

Anonymous Processors with Synchronous Shared Memory ^{*}

Bogdan S. Chlebus [†]

Gianluca De Marco [‡]

Muhammed Talo [†]

Abstract

We consider synchronous distributed systems in which anonymous processors communicate by shared read-write variables. The goal is to have all the processors assign unique names to themselves. We consider the instances of this problem determined by whether the number n is known or not, and whether concurrently attempting to write distinct values into the same memory cell is allowed or not, and whether the number of shared variables is a constant independent of n or it is unbounded. For known n , we give Las Vegas algorithms that operate in the optimum expected time, as determined by the amount of available shared memory, and use the optimum $\mathcal{O}(n \log n)$ expected number of random bits. For unknown n , we give Monte Carlo algorithms that produce correct output upon termination with probabilities that are $1 - n^{-\Omega(1)}$, which is best possible when terminating almost surely and using $\mathcal{O}(n \log n)$ random bits.

Key words: anonymous processors, synchrony, shared memory, read-write registers, naming, randomized algorithm, Las Vegas algorithm, Monte Carlo algorithm, lower bound.

^{*}The work of the first author was supported by the National Science Foundation under Grant 1016847.

[†]Department of Computer Science and Engineering, University of Colorado Denver, Denver, Colorado 80217, USA.

[‡]Dipartimento di Informatica, Università degli Studi di Salerno, Fisciano, 84084 Salerno, Italy.

1 Introduction

We consider a distributed system in which some n processors communicate using read-write shared memory. It is assumed that operations performed on shared memory occur synchronously, in that executions of algorithms are structured as sequences of globally synchronized rounds. The model of synchronous systems with read-write registers is known as the Parallel Random Access Machine (PRAM). It is a generalization of the Random Access Machine model of sequential computation [60] to the realm of synchronous concurrent processing.

We study the problem of assigning unique integer names from the interval $[1, n]$ to the n processors of a PRAM, when originally the processors do not have distinct identifiers. This task is called *naming* and is understood such that all the processors cooperate by executing a distributed algorithm to assign unique names to themselves. We assume that the original anonymous processors do not have any feature facilitating identification or distinguishing one from another. When processors of a distributed/parallel system are anonymous then the task of assigning unique identifiers to all processors is a key step in making the system fully operational, because names are needed for executing deterministic algorithms.

The task to assign unique names to anonymous processes by themselves in distributed systems can be considered as a stage in either building such systems or making them fully operational. Correspondingly, this may be categorized as either an architectural challenge or an algorithmic one. For example, tightly synchronized message passing systems are typically considered under the assumption that processors are already equipped with unique identifiers. This is because such systems impose strong demands on the architecture and the task of assigning identifiers to processors is modest when compared to providing synchrony. Similarly, when synchronous parallel machines are designed, then processors may be identified by how they are attached to the underlying communication network. In contrast to that, PRAM is a virtual model in which processors communicate via shared memory; see an exposition of PRAM as a programming environment given by Keller et al. [41]. This model does not assume any relation between the shared memory and the processors that would be conducive to identifying processors.

Distributed systems with shared read-write registers are usually considered to be asynchronous. Synchrony in such environments can be added by simulation rather than by a supportive architecture or an underlying communication network. Processes do not need to be hardware nodes, instead, they can be virtual computing agents. When a synchronous PRAM is considered, as obtained by a simulation, then the underlying system architecture does not facilitate identifying processors, and so we do not necessarily expect that processors are equipped with distinct identifiers at the start of a simulation.

We view PRAM as an abstract construct which provides a distributed environment to develop algorithms with multiple agents/processors working concurrently; see Vishkin [61] for a comprehensive exposition of PRAM as a vehicle facilitating parallel programming and harnessing the power of multi-core computer architectures. Assigning names to processors by themselves in a distributed manner is a plausible stage in an algorithmic development of such environments, as it cannot be delegated to the stage of building hardware of a parallel machine.

We consider two categories of naming problems depending on how much shared memory is available for a PRAM. In one case, the memory is bounded, in that just a constant number of memory cells is available. This means that the amount of memory is independent from the number

PRAM Model	Memory	Time	Algorithm
Arbitrary	$\mathcal{O}(1)$	$\mathcal{O}(n)$	ARBITRARY-BOUNDED-LV in Section 4
Arbitrary	$\mathcal{O}(n/\log n)$	$\mathcal{O}(\log n)$	ARBITRARY-UNBOUNDED-LV in Section 5
Common	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	COMMON-BOUNDED-LV in Section 6
Common	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	COMMON-UNBOUNDED-LV in Section 7

Table 1: Four naming problems for known n , as determined by the PRAM model and the available amount of shared memory, with the respective performance bounds. All algorithms are Las Vegas.

of processors n but as large as needed in an algorithm’s design. In the other case, the number of shared memory cells is unbounded, in the sense that it is unlimited in principle but how much is actually used by an algorithm depends on n . When it is assumed that an unbounded amount of memory cells is available, then the expected number of memory cells that are actually used may be considered as a performance metric.

Independently of the amount of shared memory available, we consider two versions of the naming problem, determined by the semantics of concurrent writing in the underlying model of computation. This is represented by the corresponding PRAM variants, which are either the Arbitrary PRAM or the Common PRAM. The Arbitrary PRAM allows to attempt to write concurrently distinct values into a register, and an arbitrary one of them gets written. The Common PRAM variant allows only equal values to be concurrently written into a register.

Randomized algorithms are typically categorized as either Las Vegas or Monte Carlo; this categorization is understood as follows. A randomized algorithm is Las Vegas when it terminates almost surely and the algorithm returns a correct output upon termination. A randomized algorithm is Monte Carlo when it terminates almost surely and an incorrect output may be produced upon termination, but the probability of error converges to zero with the number of processors growing unbounded.

We say that a parameter of an algorithmic problem is *known* when it can be used in a code of an algorithm. When the number of processors n is known, then we give Las Vegas algorithms for each of the four cases of naming determined by the kind of PRAM model and the amount of shared memory. When the number of processors is unknown, then we give Monte Carlo algorithms for each of the respective four cases of naming.

The summary of the results. We consider randomized algorithms executed by anonymous processors that operate in a synchronous manner using read-write shared memory with the goal to assign unique names to the processors. The algorithms have to be randomized (no deterministic exist) and when the number of processors is unknown then they need to be Monte Carlo (no Las Vegas exist).

We show that naming algorithms for n processors using $C > 0$ shared memory cells need to operate in $\Omega(n/C)$ expected time on an Arbitrary PRAM, and in $\Omega(n \log n/C)$ expected time on a Common PRAM. We prove additionally that any naming algorithm needs to work in the expected

PRAM Model	Memory	Time	Algorithm
Arbitrary	$\mathcal{O}(1)$	$\mathcal{O}(n)$	ARBITRARY-BOUNDED-MC in Section 8
Arbitrary	unbounded	polylog	ARBITRARY-UNBOUNDED-MC in Section 9
Common	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	COMMON-BOUNDED-MC in Section 10
Common	unbounded	polylog	COMMON-UNBOUNDED-MC in Section 11

Table 2: Four naming problems for unknown n , as determined by the PRAM model and the available amount of shared memory, with the respective performance bounds. All the algorithms are Monte Carlo. When time is marked as “polylog” then this means that the algorithm comes in two variants, such that in one the expected time is $\mathcal{O}(\log n)$ and the amount of used shared memory is suboptimal $n^{\mathcal{O}(1)}$, and in the other the expected time is suboptimal $\mathcal{O}(\log^2 n)$ but the amount of used shared memory misses optimality only by at most a logarithmic factor.

time $\Omega(\log n)$; this bound is relevant only when there is unbounded supply of shared memory. We show that, for unknown n , a Monte Carlo naming algorithm that uses $\mathcal{O}(n \log n)$ random bits has to fail to assign unique names with probability that is $n^{-\Omega(1)}$.

We consider eight specific naming problems for PRAMs. They are determined by the following independent specifications: whether n is known or not, what is the amount of shared memory (constant versus unbounded), and by the PRAM variant (Arbitrary versus Common).

For the case of known n , the naming algorithms we give are all Las Vegas. The naming problems’ specifications and the corresponding algorithms with their performance bounds are summarized in Table 1. These algorithms operate in asymptotically optimal times, for a given amount of shared memory, and use the optimum expected number $\mathcal{O}(n \log n)$ of random bits. When the amount of memory is unbounded, they use only the amount of space that is provably necessary to attain their running-time performance.

For the case of unknown n , the naming algorithms we give are all Monte Carlo. The list of the naming problems’ specifications and the corresponding algorithms with their performance bounds are summarized in Table 2. All Monte Carlo algorithms that we give have the polynomial probability of error, which is best possible when using the $\mathcal{O}(n \log n)$ expected number of random bits. When the shared memory is bounded, then these algorithms operate in asymptotically optimal times, for bounded memory, and use the optimum expected number $\mathcal{O}(n \log n)$ of random bits. When there is unbounded supply of shared memory, then we give two variants of the algorithms for Arbitrary PRAM and two for Common PRAM, with the goal to optimize different performance metrics. The set of integers used as names always makes a contiguous segment starting from the smallest name 1, so that the only possible kind of error is in assigning duplicate names.

Previous and related work. The naming problem for a synchronous PRAM has not been previously considered in the literature, to the best of the authors’ knowledge. There is a voluminous literature on various aspects of computing and communication in anonymous systems, the following review is necessarily selective.

We begin with the previous work on naming in shared-memory systems with read-write registers. A systematic exposition of shared-memory algorithm can be found in [14], when approached from the distributed-computing perspective, and in [39], when approached from the parallel-computing one.

Lipton and Park [48] considered naming in asynchronous distributed systems with read-write shared memory controlled by adaptive schedulers; they proposed a solution that terminates with positive probability, and which can be made arbitrarily close to 1 assuming that n is known. Egecioglu and Singh [28] proposed a polynomial-time Las Vegas naming algorithm for asynchronous systems with known n and read-write shared memory with oblivious scheduling of events. Kutten et al. [47] provided a thorough study of naming in asynchronous systems of shared read-write memory. They gave a Las Vegas algorithm for an oblivious scheduler for the case of known n , which works in the expected time $\mathcal{O}(\log n)$ while using $\mathcal{O}(n)$ shared registers, and also showed that a logarithmic time is required to assign names to anonymous processes. Additionally, they showed that if n is unknown then a Las Vegas naming algorithm does not exist, and a finite-state Las Vegas naming algorithm can work only for an oblivious scheduler. Panconesi et al. [54] gave a randomized wait-free naming algorithm for anonymous systems with processes prone to crashes that communicate by single-writer registers. The model considered in that work assigns unique single-writer registers to nameless processes and so has a potential to defy the impossibility of wait-free naming for general multi-writer registers, that impossibility proved by Kutten et al. [47]. Buhrman et al. [18] considered the relative complexity of naming and consensus problems in asynchronous systems with shared memory that are prone to crash failures, demonstrating that naming is harder than consensus.

Next, we review work on problems in anonymous distributed systems different from naming. Aspnes et al. [7] gave a comparative study of anonymous distributed systems with different communication mechanisms, including broadcast and shared-memory objects of various functionalities, like read-write registers and counters. Alistarh et al. [3] gave randomized renaming algorithms that act like naming ones, in that process identifiers are not referred to; for more on renaming see [2, 9, 22]. Aspnes et al. [8] considered solving consensus in anonymous systems with infinitely many processes. Attiya et al. [11] and Jayanti and Toueg [40] studied the impact of initialization of shared registers on solvability of tasks like consensus and wakeup in fault-free anonymous systems. Bonnet et al. [17] considered solvability of consensus in anonymous systems with processes prone to crashes but augmented with failure detectors. Guerraoui and Ruppert [37] showed that certain tasks like time-stamping, snapshots and consensus have deterministic solutions in anonymous systems with shared read-write registers and with processes prone to crashes. Ruppert [57] studied the impact of anonymity of processes on wait-free computing and mutual implementability of types of shared objects.

The problem of concurrent communication in anonymous networks was first considered by Angluin [4]. That work showed, in particular, that randomization was needed in naming algorithms when executed in environments that are perfectly symmetric; other related impossibility results were surveyed by Fich and Ruppert [32].

The work about anonymous networks that followed was either on specific network topologies or on problems in general message-passing systems. Most popular specific topologies included that of a ring and hypercube. In particular, the ring topology was investigated by Attiya et al. [12, 13], Flocchini et al. [33], Diks et al. [27], Itai and Rodeh [38], and Kranakis et al. [44], and the hypercube topology was studied by Kranakis and Krizanc [43] and Kranakis and Santoro [46].

The work on algorithmic problems in anonymous networks of general topologies or anonymous/named agents in anonymous/named networks included the following contributions. Afek and Matias [1] and Schieber and Snir [59] considered leader election, finding spanning trees and naming in general anonymous networks. Angluin et al. [5] studied adversarial communication by anonymous agents and Angluin et al. [6] considered self-stabilizing protocols for anonymous asynchronous agents deployed in a network of unknown size. Chalopin et al. [19] studied naming and leader election in asynchronous networks when a node knows the map of the network but its position on the map is unknown. Chlebus et al. [20] considered assigning names to anonymous stations attached to a channel that allows only beeps to be heard. Chlebus et al. [21] investigated anonymous complete networks whose links and nodes are subject to random independent failures in which single fault-free node has to wake up all nodes by propagating a wakeup message through the network. Dereniowski and Pelc [25] considered leader election among anonymous agents in anonymous networks. Dieudonné and Pec [26] studied teams of anonymous mobile agents in networks that execute deterministic algorithm with the goal to convene at one node. Fraigniaud et al. [34] considered naming in anonymous networks with one node distinguished as leader. Gašieniec et al. [35] investigated anonymous agents pursuing the goal to meet at a node or edge of a ring. Glacet et al. [36] considered leader election in anonymous trees. Kowalski and Malinowski [42] studied named agents meeting in anonymous networks. Kranakis et al. [45] investigated computing boolean functions on anonymous networks. Métivier et al. [50] considered naming anonymous unknown graphs. Michail et al. [51] studied the problems of naming and counting nodes in dynamic anonymous networks. Pelc [55] considered activating an anonymous ad hoc radio network from a single source by a deterministic algorithm. Yamashita and Kameda [62] investigated topological properties of anonymous networks that are conducive to have deterministic solutions for representative algorithmic problems.

General questions of computability in anonymous message-passing systems implemented in networks were studied by Boldi and Vigna [16], Emek et al. [29], and Sakamoto [58].

Lower bounds on PRAM were given by Fich et al. [31], Cook et al. [23], and Beame [15], among others. A review of lower bounds based on information-theoretic approach is given by Attiya and Ellen [10]. Yao's minimax principle was given by Yao [63]; the book by Motwani and Raghavan [53] gives examples of applications.

2 Technical Preliminaries

A distributed system with shared memory in which some n processors operate concurrently is the model of computation that we use in this paper. The essential properties of such systems that we assume are, first, that shared memory cells have only reading/writing capabilities, and, second, that operations of accessing shared registers are globally synchronized so that processors work in lockstep.

An execution of a synchronous algorithm is structured as a sequence of *rounds* so that each processor performs either a read from a shared memory cell or a write to a shared memory cell in a round. We assume that a processor carries out its private computation in a round in a negligible portion of the round. An invocation of either reading from or writing to a memory location is completed in the round of invocation. This model of computation is referred to in the literature as the *Parallel Random Access Machine (PRAM)*; see [39, 56].

PRAM is usually defined as a model with unbounded number of shared-memory cells, by analogy

with the random-access machine (RAM) model for sequential computing [60]. In this paper, we consider the following two instantiations of the PRAM model, determined by the amount of the available shared memory. In one situation, there is a constant number of shared memory cells, which is independent of the number of processors n but as large as needed in the specific algorithm. In the other case, the number of shared memory cells is unbounded in principle, but the expected number of shared registers accessed in an execution depends on n and is sought to be minimized.

Each shared memory cell is assumed to be initialized to 0 as a default value. This assumption simplifies the exposition, but it is not crucial as any algorithm assuming such an initialization can be modified to work with dirty memory; for example, one can apply an approach similar to that in [47]. A shared memory cell can store any value as needed in algorithms, in particular, integers of magnitude that may depend on n ; all our algorithms require a memory cell to store $\mathcal{O}(\log n)$ bits. Processors can generate as many private random bits per round as needed; all these random bits generated in an execution are assumed to be independent.

PRAM variants. Two operations are said to be performed concurrently when they are invoked in the same round of an execution of an algorithm on a PRAM. A *concurrent read* occurs when a group of processors read from the same memory cell in the same round; this results in each of these processors obtaining the value stored in the memory cell at the end of the preceding round. A *concurrent write* occurs when a group of processors invoke a write to the same memory cell in the same round.

Without loss of generality, we may assume that a concurrent read of a memory cell and a concurrent write to the same memory cell do not occur simultaneously: this is because we could designate rounds only for reading and only for writing depending on their parity, thereby slowing the algorithm by a factor of two.

The meaning of concurrent reading from the same memory cell is straightforward, in that all the readers get the value stored in this memory cell. We need to specify which value gets written to a memory cell in a concurrent write, when multiple distinct values are attempted to be written. Such stipulations determine the corresponding variants of the model. We will consider algorithms for the following two PRAM variants determined by their respective concurrent-write semantics.

Common PRAM is defined by the property that when a group of processors want to write to the same shared memory cell in a round then all the values that any of the processors want to write must be identical, otherwise the operation is illegal. Concurrent attempts to write the same value to a memory cell result in this value getting written in this round.

Arbitrary PRAM allows attempts to write any legitimate values to the same memory cell in the same round. When this occurs, then one of these values gets written, while a selection of this written value is arbitrary. All possible selections of values that get written need to be taken into account when arguing about correctness of an algorithm.

We will rely on certain standard algorithms developed for PRAMs, as explained in [39, 56]. One of them is for prefix-type computations. A typical situation in which it is applied occurs when there is an array of m shared memory cells, each memory cell storing either 0 or 1. This may represent an array of bins where 1 stands for a nonempty bin while 0 for an empty bin. Let the rank of a nonempty bin of address x be the number of nonempty bins with addresses smaller than or equal to x . Ranks can be computed in time $\mathcal{O}(\log m)$ by using an auxiliary memory of $\mathcal{O}(m)$

cells, assuming there is at least one processor assigned to each nonempty bin, while other processors do not participate; such assignment for anonymous processors will be determined by writes they performed to the bins. The underlying idea is that bins are associated with the leaves of a binary tree. The processors traverse a binary tree from the leaves to the root and back to the leaves. When updating information at a node, only the information stored at the parent, the sibling and the children is used.

We may observe that the same memory can be used repeatedly when such computation needs to be performed multiple times on the same tree. A possible approach is to verify if the information at a needed memory cell, representing either a parent, a sibling or a child of a visited node, is fresh or rather stale from previous executions. This could be accomplished in the following three steps by a processor. First, the processor erases a memory cell it needs to read by rewriting its present value by a blank value. Second, the processor writes again the value at the node it is visiting, which may have been erased in the previous step by other processors that need the value. Finally, the processor reads again the memory cell it has just erased, to see if it stays erased, which means its contents were stale, or not, which in turn means its contents got rewritten so they are fresh.

Balls into bins. In the course of probabilistic analysis of algorithms, we will often model actions of processors by throwing balls into bins. This can be done in two natural ways. One is such that memory addresses are interpreted as bins and the values written represent balls, possibly with labels. Then the total number of balls considered will always be n , that is, equal to the number of processors of a PRAM. Another possibility is when bins represent rounds and selecting a bin results in performing a write to a suitable shared register in the respective round.

The following terms refer to the status of a bin in a given round. A bin is called *empty* when there are no balls in it. A ball is *single* in a bin when there are no other balls in the same bin, and such a bin can be called *singleton*. A bin is *multiple* when there are at least two balls in it. A *collision* occurs in a multiple bin. Finally, a bin with at least one ball is *occupied*.

The *rank* of a bin containing a ball is the number of bins with smaller or equal names that contain balls. When each processor, in a group of processors that still seek names, throws a ball and there is no collision then this breaks symmetry in a manner that in principle could facilitate assigning unique names to processors in the group, namely, the ranks of selected bins may serve as names.

Throwing balls into bins will be performed repeatedly in each instance of modeling the behavior of an algorithm. Each instance of throwing a number of balls into bins is then called a *stage*. There will be an additional numeric parameter $\beta > 0$, and we call the process of throwing balls into bins the β -*process*, accordingly. This parameter β may determine the number of bins in a stage and also when a stage is the last one in an execution of the β -process. The specifications of any such a β -process apply only within the section in which it is defined.

In a given stage of a β -process, the balls that are thrown into the bins are called *eligible* for the stage; all the n balls are eligible for the first stage. When a bin is selected for a ball to be placed in, then this occurs uniformly at random over the range of bins, and independently over the considered balls. Each selection of a bin for an eligible balls requires the number of random bits equal to the binary logarithm of the number of the available bins.

When we sum up the numbers of available bins over all the stages of an execution of a β -process until termination, then the result is *the number of bins ever needed* in this execution. Similarly,

Procedure VERIFY-COLLISION(x)

```
initialize Heads[ $x$ ]  $\leftarrow$  Tails[ $x$ ]  $\leftarrow$  false  
toss $_v \leftarrow$  outcome of tossing a fair coin  
if toss $_v$  = tails  
    then Tails[ $x$ ]  $\leftarrow$  true  
    else Heads[ $x$ ]  $\leftarrow$  true  
return (Tails[ $x$ ] = Heads[ $x$ ])
```

Figure 1: A pseudocode for a processor v of a Common PRAM, where x is a positive integer. **Heads** and **Tails** are arrays of shared memory cells. When the parameter x is dropped in a call then this means that $x = 1$. The procedure returns **true** when a collision is detected.

the number of bits ever generated in an execution of a β -process is the sum of all the numbers of random bits needed to be generated to place balls, over all the stages and balls until termination of this execution.

The idea of representing attempts to assign names as throwing balls into bins is quite generic. In particular, it was applied by Egecioğlu and Singh [28], who proposed a synchronous algorithm that repeatedly throws all balls together into all available bins, the selections of bins for balls made independently and uniformly at random. In their algorithm for n processors, we can use $\gamma \cdot n$ memory cells, where $\gamma > 1$. Let us choose $\gamma = 3$ for the following calculations to be specific. This algorithm has an exponential expected-time performance. To see this, we estimate the probability that each bin is either singleton or empty. Let the balls be thrown one by one. After the first $n/2$ balls are in singleton bins, the probability to hit an empty bin is at most $\frac{2.5n}{3n} = \frac{5}{6}$; we treat this as a success in a Bernoulli trial. The probability of $n/2$ such successes is at most $(\frac{5}{6})^{n/2}$, so the expected time to wait for the algorithm to terminate is at least $(\sqrt{\frac{6}{5}})^n$, which is exponential in n .

We consider related processes that could be as fast as $\mathcal{O}(\log n)$ in expected time, while still using only $\mathcal{O}(n)$ shared memory cells, see Section 7. The idea is to let balls in singleton bins stay put and only move those that collided with other balls by landing in bins that became thereby multiple. To implement this on a Common PRAM, we need a way to detect collisions, which we explain next.

Verifying collisions. We will use a randomized procedure for Common PRAM to verify if a collision occurs in a bin, say, a bin x , which is executed by each processor that selected bin x . This procedure VERIFY-COLLISION is represented in Figure 1. There are two arrays **Tails** and **Heads** of shared memory cells. Bin x is verified by using memory cells **Tails**[x] and **Heads**[x]. First, the memory cells **Tails**[x] and **Heads**[x] are set each to false. Next, each processors selects randomly and independently one of these memory cells and sets it to true. Finally, every processor reads both **Tails**[x] and **Heads**[x] and detects a collision upon reading **true** twice.

Lemma 1 *For an integer x , procedure $\text{VERIFY-COLLISION}(x)$ executed by one processor never detects a collision, and when multiple processors execute this procedure then a collision is detected with probability at least $\frac{1}{2}$.*

Proof: When only one processor executes the procedure, then first the processor sets both $\text{Heads}[x]$ and $\text{Tails}[x]$ to false and next only one of them to true. This guarantees that $\text{Heads}[x]$ and $\text{Tails}[x]$ store different values and so collision is not detected. When some $m > 1$ processors execute the procedure, then collision is not detected only when either all processors set $\text{Heads}[x]$ to true or all processors set $\text{Tails}[x]$ to true. This means that the processors generate the same outcome in their coin tosses. This occurs with probability 2^{-m+1} , which is at most $\frac{1}{2}$. \square

Pseudocode conventions and notations. We specify algorithms using pseudocode conventions natural for the PRAM model. An example of such a representation is in Figure 1. These conventions are summarized as follows.

There are two kinds of variables: shared and private. The names of shared variables start with capital letters and the names of private ones are all in small letters. To emphasize that a private variable x is such that its value may depend on a processor v in a round, we may denote x by x_v in a pseudocode for v .

When x is a private variable that may have different values at different processors at the same time, then we denote this variable used by a processor v by x_v . Private variables that have the same value at the same time in all the processors are usually used without subscripts, like variables controlling for-loops. An assignment instruction of the form $x \leftarrow y \leftarrow \dots \leftarrow z \leftarrow \alpha$, where x, y, \dots, z are variables and α is a value, means to assign α as the value to be stored in all the listed variables x, y, \dots, z .

We want that, at any round of an execution, all the processors that have not terminated yet are executing the same line of the pseudocode. In particular, when an instruction is conditional on a statement then a processor that does not meet the condition pauses as long as it would be needed for all the processors that meet the condition complete their instructions, even when there are no such processors. If this is not a constant-time instruction, then it may incur an unnecessary time cost. To avoid this problem, we may first verify if there is some processor that satisfies the condition, which can be done in constant time.

We use three notations for logarithms. The notation $\lg x$ stands for the logarithm of x to the base 2. The notation $\ln x$ denotes the natural logarithm of x . When the base of logarithms does not matter then we use $\log x$, like in the asymptotic notation $\mathcal{O}(\log x)$.

Properties of naming algorithms. Naming algorithms in distributed environments involving multi-writer read-write shared memory have to be randomized to break symmetry [4, 14]. An eventual assignment of proper names cannot be a sure event, because, in principle, two processors can generate the same strings of random bits in the course of an execution. We say that an event *is almost sure*, or *occurs almost surely*, when it occurs with probability 1. When n processors generate their private strings of random bits then it is an almost sure event that all these strings are eventually pairwise distinct. Therefore, a most advantageous scenario that we could expect, when a set of n processors is to execute a randomized naming algorithm, is that the algorithm eventually terminates almost surely and that at the moment of termination the output is *correct*, in that the assigned names are without duplicates and fill the whole interval $[1, n]$.

Randomized naming algorithms are categorized as either Monte Carlo or Las Vegas, which are defined as follows. A randomized algorithm is *Las Vegas* when it terminates almost surely and the algorithm returns a correct output upon termination. A randomized algorithm is *Monte Carlo* when it terminates almost surely and an incorrect output may be produced upon termination, but the probability of error converges to zero with the size of input growing unbounded.

We give algorithms that use the expected number of $\mathcal{O}(n \log n)$ random bits with a large probability. This amount of random information is necessary if an algorithm is to terminate almost surely. The following fact is essentially a folklore, but since we do not know if it was ever proved in the literature, we give a proof for completeness' sake. Our arguments resort to the notions of information theory [24].

Proposition 1 *If a randomized naming algorithm is correct with probability p_n , when executed by n anonymous processors, then it requires $\Omega(n \log n)$ random bits with probability at least p_n . In particular, a Las Vegas naming algorithm for n processors uses $\Omega(n \log n)$ random bits almost surely.*

Proof: Let us assign conceptual identifiers to the processors, for the sake of argument. These *unknown identifiers* are known only to an external observer and not to algorithms. The purpose of executing the algorithm is to assign explicit identifiers, which we call *given names*.

Let a processor with an unknown identifier u_i generate a string of bits b_i , for $i = 1, \dots, n$. A distribution of given names among the n anonymous processors, which results from executing the algorithm, is a random variable X_n with a uniform distribution on the set of all permutations of the unknown identifiers. This is because of symmetry: all processors execute the same code, without explicit private names, and if we rearrange the strings generated bits b_i among the processors u_i , then this results in the corresponding rearrangement of the given names.

The underlying probability space consists of $n!$ elementary events, each determined by an assignment of the given names to the processors identified by the unknown identifiers. It follows that each of these events occurs with probability $1/n!$. The Shannon entropy of the random variable X_n is thus $\lg(n!) = \Theta(n \log n)$. The decision about which assignment of given names is produced is determined by the random bits, as they are the only source of entropy. It follows that the expected number of random bits used by the algorithm needs to be as large as the entropy of the random variable X_n .

The property that all assigned names are distinct and in the interval $[1, n]$ holds with probability p_n . An execution needs to generate a total of $\Omega(n \log n)$ random bits with probability at least p_n , because of the bound on entropy. A Las Vegas algorithm terminates almost surely, and returns correct names upon termination. This means that $p_n = 1$ and so that $\Omega(n \log n)$ random bits are used almost surely. \square

A naming algorithm cannot be Las Vegas when n is unknown, as was observed by Kutten et al. [47] for asynchronous computations against an oblivious adversary. We show the analogous fact for synchronous computations.

Proposition 2 *There is no Las Vegas naming algorithm for a PRAM with $n > 1$ processors that does not refer to the number of processors n in its code.*

Proof: Let us suppose, to arrive at a contradiction, that such a naming Las Vegas algorithm exists. Consider a system of $n - 1 \geq 1$ processors, and an execution \mathcal{E} on these $n - 1$ processors that uses specific strings of random bits such that the algorithm terminates in \mathcal{E} with these random bits. Such strings of random bits exist because the algorithm terminates almost surely.

Let v_1 be a processor that halts latest in \mathcal{E} among the $n - 1$ processors. Let $\alpha_{\mathcal{E}}$ be the string of random bits generated by processor v_1 by the time it halts in \mathcal{E} . Consider the execution \mathcal{E}' on $n \geq 2$ processors such that $n - 1$ processors obtain the same strings of random bits as in \mathcal{E} and an extra processor v_2 obtains $\alpha_{\mathcal{E}}$ as its random bits. The executions \mathcal{E} and \mathcal{E}' are indistinguishable for the $n - 1$ processors participating in \mathcal{E} , so they assign themselves the same names and halt. Processor v_2 performs the same reads and writes as processor v_1 and assigns itself the same name as processor v_1 does and halts in the same round as processor v_1 . This is the termination round because by that time all the other processor have halted as well.

It follows that execution \mathcal{E}' results in a name being duplicated. The probability of duplication for n processors is at least as large as the probability to generate two identical finite random strings in \mathcal{E}' for some two processors, so this probability is positive. \square

If n is unknown, then the restriction $\mathcal{O}(n \log n)$ on the number of random bits makes it inevitable that the probability of error is at least polynomially bounded from below, as we show next.

Proposition 3 *For unknown n , if a randomized naming algorithm is executed by n anonymous processors, then an execution is incorrect, in that duplicate names are assigned to distinct processors, with probability that is at least $n^{-\Omega(1)}$, assuming that the algorithm uses $\mathcal{O}(n \log n)$ random bits with probability $1 - n^{-\Omega(1)}$.*

Proof: Suppose the algorithm uses at most $cn \lg n$ random bits with probability p_n when executed by a system of n processors, for some constant $c > 0$. Then one of these processors uses at most $c \lg n$ bits with probability p_n , by the pigeonhole principle.

Consider an execution for $n+1$ processors. Let us distinguish a processor v . Consider the actions of the remaining n processors: one of them, say w , uses at most $c \lg n$ bits with the probability p_n . Processor v generates the same string of bits with probability $2^{-c \lg n} = n^{-c}$. The random bits generated by w and v are independent. Therefore duplicate names occur with probability at least $n^{-c} \cdot p_n$. When we have a bound on probabilities p_n to be $p_n = 1 - n^{-\Omega(1)}$, then probability of occurrence of duplicate names is at least $n^{-c}(1 - n^{-\Omega(1)}) = n^{-\Omega(1)}$. \square

3 Lower Bounds on Running Time

We consider two kinds of algorithmic naming problems, as determined by the amount of shared memory. One case is for a constant number of shared memory cells, for which we give an optimal lower bound on time for $\mathcal{O}(1)$ shared memory. The other case is when the number of shared memory cells and their capacity are unbounded, for which we give an “absolute” lower bound on time. We begin with lower bounds that reflect the amount of shared memory.

Intuitively, as processors generate random bits, these bits need to be made common knowledge through some implicit process that assigns explicit names. There is an underlying flow of information to spread knowledge among the processors through the available shared memory. Time is

bounded from below by the rate of flow of information and the total amount of bits that need to be shared.

On the technical level, in order to bound the expected time of a randomized algorithm, we apply the Yao's minimax principle [63] to relate this expected time to the distributional expected time complexity. A randomized algorithm whose actions are determined by random bits can be considered as a probability distribution on deterministic algorithms. A deterministic algorithm has strings of bits given to processors as their inputs, with some probability distribution on such inputs. The expected time of such a deterministic algorithm, given any specific probability distribution on the inputs, is a lower bound on the expected time of a randomized algorithm.

To make such interpretation of randomized algorithms possible, we consider strings of bits of equal length. With such a restriction on inputs, deterministic algorithm may not be able to assign proper names for some assignments of inputs, for example, when all the inputs are equal. We augment such deterministic algorithms by adding an option for the algorithm to withhold a decision on assignment of names and output "no name" for some processors. This is interpreted as the deterministic algorithm needing longer inputs, for which the given inputs are prefixes, and which for the randomized algorithm means that some processors need to generate more random bits.

Regarding probability distributions for inputs of a given length, it will always be the uniform distribution. This is because we will use an assessment of the amount of entropy of such a distribution.

Theorem 1 *A randomized naming algorithm for a Common PRAM with n processors and $C > 0$ shared memory cells operates in $\Omega(n \log n / C)$ expected time when it is either a Las Vegas algorithm or a Monte Carlo algorithm with the probability of error smaller than $1/2$.*

Proof: We consider Las Vegas algorithms in this argument, the Monte Carlo case is similar, the difference is in applying Yao's principle for Monte Carlo algorithms. We interpret a randomized algorithm as a deterministic one working with all possible assignments of random bits as inputs with a uniform mass function on the inputs. The expected time of the deterministic algorithm is a lower bound on the expected time of the randomized algorithm.

There are $n!$ possible assignments of given names to the processors. Each of them occurs with the same probability $1/n!$ when the input bit strings are assigned uniformly at random. Therefore the entropy of name assignments, interpreted as a random variable, is $\lg n! = \Omega(n \log n)$.

Next we consider executions of such a deterministic algorithm on the inputs with a uniform probability distribution. We may assume without loss of generality that an execution is structured into the following phases, each consisting of $C + 1$ rounds. In the first round of a phase, each processor either writes into a shared memory cell or pauses. In the following rounds of a phase, every processor learns the current values of each among the C memory cells. This may take C rounds for every processor to scan the whole shared memory, but we do not include this reading overhead as contributing to the lower bound. Instead, since this is a simulation anyway, we conservatively assume that the process of learning all the contents of shared memory cells at the end of a phase is instantaneous and complete.

The Common variant of PRAM requires that if a memory cell is written into concurrently then there is a common value that gets written by all the writers. Such a value needs to be determined

by the code and the address of a memory cell. This means that, for each phase and any memory cell, a processor choosing to write into this memory cell knows the common value to be written. By the structure of execution, in which all processors read all the registers after a round of writing, any processor knows what value gets written into each available memory cell in a phase, if any is written into a particular cell. This implies that the contents written into shared memory cells may not convey any new information but are already implicit in the states of the processors represented by their private memories after reading the whole shared memory.

When a processor reads all the shared memory cells in a phase, then the only new information it may learn is the addresses of memory cells into which new writes were performed and those into which there were no new writes. This makes it possible obtain at most C bits of information per phase, because each register was either written into or not.

There are $\Omega(n \log n)$ bits of information that need to be settled and one phase changes the entropy by at most C bits. It follows that the expected number of phases of the deterministic algorithm is $\Omega(n \log n / C)$. By the Yao's principle, $\Omega(n \log n / C)$ is a lower bound on the expected time of a randomized algorithm. \square

For Arbitrary PRAM, writing can spread information through the written values, because different processes can attempt to write distinct strings of bits. The rate of flow of information is constrained by the fact that when multiple writers attempt to write to the same memory cell then only one of them succeeds, if the values written are distinct. This intuitively means that the size of a group of processors writing to the same register determines how much information the writers learn by subsequent reading. These intuitions are made formal in the proof of the following Theorem 2.

Theorem 2 *A randomized naming algorithm for an Arbitrary PRAM with n processors and $C > 0$ shared memory cells operates in $\Omega(n/C)$ expected time when it is either a Las Vegas algorithm or a Monte Carlo algorithm with the probability of error smaller than $1/2$.*

Proof: We consider Las Vegas algorithms in this argument, the Monte Carlo case is similar, the difference is in applying Yao's principle for Monte Carlo algorithms. We again replace a given randomized algorithm by its deterministic version that works on assignments of strings of bits of the same length as inputs, with such inputs assigned uniformly at random to the processors. The goal is to use the property that the expected time of this deterministic algorithm, for a given probability distribution of inputs, is a lower bound on the expected time of the randomized algorithm. Next, we consider executions of this deterministic algorithm.

Similarly as in the proof of Theorem 1, we observe that there are $n!$ assignments of given names to the processors and each of them occurs with the same probability $1/n!$, when the input bit strings are assigned uniformly at random. The entropy of name assignments is again $\lg n! = \Omega(n \log n)$. The algorithm needs to make the processors learn $\Omega(n \log n)$ bits using the available $C > 0$ shared memory cells.

We may interpret an execution as structured into phases, such that each processor performs at most one write in a phase and then reads all the registers. The time of a phase is assumed conservatively to be $\mathcal{O}(1)$. Consider a register and a group of processors that attempt to write their values into this register in a phase. The values attempted to be written are represented as strings of bits. If some of these values have 0 and some have 1 at some bit position among the strings, then this bit position may convey one bit of information. The maximum amount of information is

provided by a write when the written string of bits facilitates identifying the writer by comparing its written value to the other values attempted to be written concurrently to the same memory cell. This amount is at most the binary logarithm of the size of this group of processors. Therefore, each memory cell written to in a round contributes at most $\lg n$ bits of information, because there may be at most n writers to it. Since there are C registers, the maximum number of bits of information learnt by the processors in a phase is $C \lg n$.

The entropy of the assignment of names is $\lg n! = \Omega(n \log n)$, so the expected number of phases of the deterministic algorithm is $\Omega(n \lg n / (C \lg n)) = \Omega(n/C)$. By the Yao's principle, this is also a lower bound on the expected time of a randomized algorithm. \square

Next, we consider “absolute” requirements on time for a PRAM to assign unique names to the available n processors. The generality of the lower bound we give stems from the weakness of the assumptions. First, nothing is assumed about the knowledge of n . Second, concurrent writing is not constrained in any way. Third, shared memory cells are unbounded in their number and size.

We show next in Theorem 3 that any Las Vegas naming algorithm has $\Omega(\log n)$ expected time for the synchronous schedule of events. The argument we give is in the spirit of similar arguments applied by Cook et al. [23] and Beame [15]. In an analogous manner, Kutten et al. [47] showed that any Las Vegas naming algorithm for asynchronous read-write shared memory systems has the expected time $\Omega(\log n)$ against a certain oblivious schedule. What these arguments share, along with the arguments we employ in this paper, are a formalization of the notion of flow of information during an execution of an algorithm, combined with a recursive estimate of the rate of this flow.

The relation *processor v knows processor w in round t* is defined recursively as follows. First, for any processor v , we have that v knows v in any round $t > 0$. Second, if a processor v writes to a shared memory cell R in a round t_1 and a processor w reads from R in a round $t_2 > t_1$, such that there was no other write into this memory cell after t_1 and prior to t_2 , then processor w knows in round t_2 each processor that v knows in round t_1 . Finally, the relation is the smallest transitive relation that satisfies the two postulates formulated above. This means that it is the smallest relation such that if processor v knows processor w in round t_1 and z knows v in round t_2 such that $t_2 > t_1$ then processor z knows w in round t_2 . In particular, the knowledge accumulates with time, in that if a processor v knows processor z in round t_1 and round t_2 is such that $t_2 > t_1$ then v knows z in round t_2 as well.

Lemma 2 *Let \mathcal{A} be a deterministic algorithm that assigns distinct names to the processors, with the possibility that some processors output “no name” for some inputs, when each node has an input string of bits of the same length. When algorithm \mathcal{A} terminates with proper names assigned to all the processors then each processor knows all the other processors.*

Proof: We may assume that $n > 1$ as otherwise one processors knows itself. Let us consider an assignment \mathcal{I} of inputs that results in a proper assignment of distinct names to all the processors when algorithm \mathcal{A} terminates. This implies that all the inputs in the assignment \mathcal{I} are distinct strings of bits, as otherwise some two processors, say, v and w that obtain the same input string of bits would either assign themselves the same name or declare “no name” as output.

Suppose that a processor v does not know a processor w , when v halts for inputs from \mathcal{I} . Consider an assignment of inputs \mathcal{J} which is the same as \mathcal{I} for processors different from w and such that the input of w is the same as input for v in \mathcal{I} . Then the actions of processor v would be the

same with \mathcal{J} as with \mathcal{I} , because v is not affected by the input of w , so that v would assign itself the same name with \mathcal{J} as with \mathcal{I} . But the actions of processor w would be the same in \mathcal{J} as those of v , because their input strings of bits are identical under \mathcal{J} . It follows that w would assign itself the name of v , resulting in duplicate names. This contradicts the assumption that all processors obtain unique names in the execution. \square

We will use Lemma 2 to assess running times by estimating the number of interleaved reads and writes needed for processors to get to know all the processors. The rate of learning such information may depend on time, because we do not restrict the amount of shared memory, unlike in Theorems 1 and 2. Indeed, the rate may increase exponentially, under most liberal estimates.

The following Theorem 3 holds for both Common and Arbitrary PRAMs. The argument used in the proof is general enough not to depend on any specific semantics of writing.

Theorem 3 *A randomized naming algorithm for a PRAM with n processors operates in $\Omega(\log n)$ expected time when it is either a Las Vegas algorithm or a Monte Carlo algorithm with the probability of error smaller than $1/2$.*

Proof: The argument is for a Las Vegas algorithm, the Monte Carlo case is similar. A randomized algorithm can be interpreted as a probability distribution on a finite set of deterministic algorithms. Such an interpretation works when input strings for a deterministic algorithm are of the same length. We consider all such possible lengths for deterministic algorithms, similarly as in the previous proofs of lower bounds.

Let us consider a deterministic algorithm \mathcal{A} , and let inputs be strings of bits of the same length. We may structure an execution of this algorithm \mathcal{A} into *phases* as follows. A phase consists of two rounds. In the first round of a phase, each processor either writes to a shared memory cell or pauses. In the second round of a phase, each processor either reads from a shared memory cell or pauses. Such structuring can be done without loss of generality at the expense of slowing down an execution by a factor of at most 2. Observe that the knowledge in the first round of a phase is the same as in the last round of the preceding phase.

Phases are numbered by consecutively increasing integers, starting from 1. A phase i comprised pairs of rounds $\{2i-1, 2i\}$, for integers $i \geq 1$. In particular, the first phase consists of rounds 1 and 2. We also add phase 0 that represents the knowledge before any reads or writes were performed.

We show the following invariant, for $i \geq 0$: a processor knows at most 2^i processors at the end of phase i . The proof of this invariant is by induction on i .

The base case is for $i = 0$. The invariant follows from the fact that a processor knows only one processor in phase 0, namely itself, and $2^0 = 1$.

To show the inductive step, suppose the invariant holds for a phase $i \geq 0$ and consider the next phase $i+1$. A processor v may increase its knowledge by reading in the second round of phase $i+1$. Suppose the read is from a shared memory cell R . The latest write into this memory cell occurred by the first round of phase $i+1$. This means that the processor w that wrote to R by phase $i+1$, as the last one that did write, knew at most 2^i processors in the round of writing, by the inductive assumption and the fact that what is written in phase $i+1$ was learnt by the immediately preceding phase i . Moreover, by the semantics of writing, the value written to R by w in that round removed any previous information stored in R . Processor v starts phase $i+1$ knowing at most 2^i processors,

and also learns of at most 2^i other processors by reading in phase $i + 1$, namely, those values known by the latest writer of the read contents. It follows that processor v knows at most $2^i + 2^i = 2^{i+1}$ processors by the end of phase $i + 1$.

When proper names are assigned by such a deterministic algorithm, then each processor knows every other processor, by Lemma 2. A processor knows every other processor in a phase j such that $2^j \geq n$, by the invariant just proved. Such a phase number j satisfies $j \geq \lg n$, and it takes $2 \lg n$ rounds to complete $\lg n$ phases.

Let us consider inputs strings of bits assigned to processors uniformly at random. We need to estimate the expected running time of an algorithm \mathcal{A} on such inputs. Let us observe that, in the context of interpreting deterministic executions for the sake to apply Yao's principle, terminating executions of \mathcal{A} that do not result in names assigned to all the processors could be pruned from a bound on their expected running time, because such executions are determined by bounded input strings of bits that a randomized algorithm would extend to make them sufficiently long to assign proper names. In other words, from the perspective of randomized algorithms, such prematurely ending executions do not represent real terminating ones.

The expected time of \mathcal{A} , conditional on terminating with proper names assigned, is therefore at least $2 \lg n$. We conclude, by the Yao's principle, that any randomized naming algorithm has $\Omega(\log n)$ expected runtime. \square

The three lower bounds on time given in this Section may be applied in two ways. One is to infer optimality of time for a given amount of shared memory used. Another is to infer optimality of shared memory use given a time performance. This is summarized in the following Corollary 1.

Corollary 1 *If the expected time of a naming Las Vegas algorithm is $\mathcal{O}(n)$ on an Arbitrary PRAM with $\mathcal{O}(1)$ shared memory, then this time performance is asymptotically optimal. If the expected time of a naming Las Vegas algorithm is $\mathcal{O}(n \log n)$ on a Common PRAM with $\mathcal{O}(1)$ shared memory, then this time performance is asymptotically optimal. If a Las Vegas naming algorithm operates in time $\mathcal{O}(\log n)$ on an Arbitrary PRAM using $\mathcal{O}(n/\log n)$ shared memory cells, then this amount of shared memory is asymptotically optimal. If a Las Vegas naming algorithm operates in time $\mathcal{O}(\log n)$ on a Common PRAM using $\mathcal{O}(n)$ shared memory cells, then this amount of shared memory is optimal.*

Proof: We verify that the lower bounds match the assumed upper bounds. By Theorem 2, a Las Vegas algorithm operates almost surely in $\Omega(n)$ time on an Arbitrary PRAM when space is $\mathcal{O}(1)$. By Theorem 1, a Las Vegas algorithm operates almost surely in $\Omega(n \log n)$ time on a Common PRAM when space is $\mathcal{O}(1)$. By Theorem 2, a Las Vegas algorithm operates almost surely in $\Omega(\log n)$ time on an Arbitrary PRAM when space is $\mathcal{O}(n/\log n)$. By Theorem 1, a Las Vegas algorithm operates almost surely in $\Omega(\log n)$ time on a Common PRAM when space is $\mathcal{O}(n)$. \square

4 Las Vegas for Arbitrary with Bounded Memory

We present a Las Vegas naming algorithm for an Arbitrary PRAM with a constant number of shared memory cells, in the case when the number of processors n is known.

During an execution of this algorithm, processors repeatedly write random strings of bits representing integers to a shared memory cell called `Pad`, and next read `Pad` to verify the outcome of

Algorithm ARBITRARY-BOUNDED-LV

```
repeat
  initialize Counter  $\leftarrow$  namev  $\leftarrow$  0
  binv  $\leftarrow$  random integer in  $[1, n^\beta]$ 
  for  $i \leftarrow 1$  to  $n$  do
    if namev = 0 then
      Pad  $\leftarrow$  binv
      if Pad = binv then
        Counter  $\leftarrow$  Counter + 1
        namev  $\leftarrow$  Counter
until Counter =  $n$ 
```

Figure 2: A pseudocode for a processor v of an Arbitrary PRAM, where the number of shared memory cells is a constant independent of n . The variables **Counter** and **Pad** are shared. The private variable **name** stores the acquired name. The constant $\beta > 0$ is parameter to be determined by analysis.

writing. A processor v that reads the same value as it attempted to write increments the integer stored in a shared register **Counter** and uses the obtained number as a tentative name, which it stores in a private variable **name_v**. The values of **Counter** could get incremented a total of less than n times, which occurs when some two processors chose the same random integer to write to the register **Pad**. The correctness of the assigned names is verified by the equality **Counter** = n , because **Counter** was initialized to zero. When such a verification fails then this results in another iteration of a series of writes to register **Pad**, otherwise the execution terminates and the value stored at **name_v** becomes the final name of processor v .

This algorithm is called ARBITRARY-BOUNDED-LV and its pseudocode is given in Figure 2. The pseudocode refers to a constant $\beta > 0$ which determines the bounded range $[1, n^\beta]$ from which processors select integers to write to the shared register **Pad**.

Balls into bins. The selection of random integers in the range $[1, n^\beta]$ by n processors can be interpreted as throwing n balls into n^β bins, which we call β -process. A collision represents two processors assigning themselves the same name. Therefore an execution of the algorithm can be interpreted as performing such ball placements repeatedly until there is no collision.

Lemma 3 *For each $a > 0$ there exists $\beta > 0$ such that when n balls are thrown into n^β bins during the β -process then the probability of a collision is at most n^{-a} .*

Proof: Consider the balls thrown one by one. When a ball is thrown, then at most n bins are already occupied, so the probability of the ball ending in an occupied bin is at most $n/n^\beta = n^{-\beta+1}$.

No collisions occur with probability that is at least

$$\left(1 - \frac{1}{n^{\beta-1}}\right)^n \geq 1 - \frac{n}{n^{\beta-1}} = 1 - n^{-\beta+2}, \quad (1)$$

by the Bernoulli's inequality. If we take $\beta \geq a + 2$ then just one iteration of the repeat-loop is sufficient with probability that is at least $1 - n^{-a}$. \square

Next we summarize the performance of algorithm ARBITRARY-BOUNDED-LV as a Las Vegas algorithm.

Theorem 4 *Algorithm ARBITRARY-BOUNDED-LV terminates almost surely and there is no error when it terminates. For any $a > 0$, there exist $\beta > 0$ and $c > 0$ and such that the algorithm terminates within time cn using at most $cn \ln n$ random bits with probability at least $1 - n^{-a}$.*

Proof: The algorithm assigns consecutive names from a continuous interval starting from 1, by the pseudocode in Figure 2. It terminates after n different tentative names have been assigned, by the condition controlling the repeat loop in the pseudocode of Figure 2. This means that proper names have been assigned when the algorithm terminates.

We map an execution of the β -process on an execution of algorithm ARBITRARY-BOUNDED-LV in a natural manner. Under such an interpretation, Lemma 3 estimates the probability of the event that the n processors select different numbers in the interval $[1, n^\beta]$ as their values to write to `Pad` in one iteration of the repeat-loop. This implies that just one iteration of the repeat-loop is sufficient with the probability that is at least $1 - n^{-a}$. The probability of the event that i iterations are not sufficient to terminate is at most n^{-ia} , which converges to 0 as i increases, so the algorithm terminates almost surely. One iteration of the repeat-loop takes $\mathcal{O}(n)$ rounds and it requires $\mathcal{O}(n \log n)$ random bits. \square

Algorithm ARBITRARY-BOUNDED-LV is optimal among Las Vegas naming algorithms with respect to its expected running time $\mathcal{O}(n)$, given the amount $\mathcal{O}(1)$ of its available shared memory, by Corollary 1 in Section 3, and the expected number of random bits $\mathcal{O}(n \log n)$, by Proposition 1 in Section 2.

5 Las Vegas for Arbitrary with Unbounded Memory

In this section, we give a Las Vegas algorithm for an Arbitrary PRAM with an unbounded supply of shared memory cells, in the case when the number of processors n is known. This algorithm is called ARBITRARY-UNBOUNDED-LV and its pseudocode is given in Figure 3.

The algorithm uses two arrays `Bin` and `Counter` of $\frac{n}{\ln n}$ shared memory cells each. An execution proceeds by repeated attempts to assign names. During each such an attempt, the processors work to assign tentative names. Next, the number of distinct tentative names is obtained and if the count equals n then the tentative names become final, otherwise another attempt is made. We assume that each such an attempt uses a new segment of memory cells `Counter` initialized to 0s, which is to simplify the exposition and analysis. An attempt to assign tentative names proceeds by each processor v selecting two integers `binv` and `labelv` uniformly at random, where `binv` $\in [1, \frac{n}{\ln n}]$ and `labelv` $\in [1, n^\beta]$.

Algorithm ARBITRARY-UNBOUNDED-LV

```
repeat
  allocate Counter[1,  $\frac{n}{\ln n}$ ]    /* array of fresh memory cells initialized to 0s */
  initialize positionv ← (0, 0)
  binv ← a random integer in [1,  $\frac{n}{\ln n}$ ]
  labelv ← a random integer in [1,  $n^\beta$ ]
  repeat
    initialize All-Named ← true
    if positionv = (0, 0) then
      Bin[binv] ← labelv
      if Bin[binv] = labelv then
        Counter[binv] ← Counter[binv] + 1
        positionv ← (binv, Counter[binv])
      else All-Named ← false
  until All-Named          /* each processor has a tentative name */
  namev ← rank of positionv
until n is the maximum name /* no duplicates among tentative names */
```

Figure 3: A pseudocode for a processor v of an Arbitrary PRAM, where the number of shared memory cells is unbounded. The variables **Bin** and **Counter** denote arrays of $\frac{n}{\ln n}$ shared memory cells each, the variable **All-Named** is also shared. The private variable **name** stores the acquired name. The constant $\beta > 0$ is a parameter to be determined by analysis.

Next the processors repeatedly attempt to write **label_v** into **Bin[bin_v]**. Each such a write is followed by a read and the lucky writer uses the value of memory register **Counter[bin_v]** to create a pair of numbers **(bin_v, Counter[bin_v])**, after first incrementing **Counter[bin_v]**, which is called **bin_v's position** and is stored in variable **position_v**. After all processors have their positions determined, we define their ranks as follows. To find the *rank* of **position_v**, we arrange all such pairs in lexicographic order, comparing first on **bin** and then on **Counter[bin]**, and the rank is the position of this entry in the resulting list, where the first entry has position 1, the second 2, and so on.

Ranks can be computed using a prefix-type algorithm operating in time $\mathcal{O}(\log n)$. This algorithm first finds for each **bin** $\in [1, \frac{n}{\ln n}]$ the sum $s(\text{bin}) = \sum_{1 \leq i < \text{bin}} \text{Counter}[i]$. Next, each processor v with a position **(bin_v, c)** assigns to itself $s(\text{bin}_v) + c$ as its rank. After ranks have been computed, they are used as tentative names.

In the analysis of algorithm ARBITRARY-UNBOUNDED-LV we will refer to the following bound on independent Bernoulli trials. Let S_n be the number of successes in n independent Bernoulli trials, with p as the probability of success. Let $b(i; n, p)$ be the probability of an occurrence of

exactly i successes. For $r > np$, the following bound holds

$$\Pr(S_n \geq r) \leq b(r; n, p) \cdot \frac{r(1-p)}{r-np}, \quad (2)$$

see Feller [30].

Balls into bins. We consider a process of throwing n balls into $\frac{n}{\ln n}$ bins. Each ball has a label assigned randomly from the range $[1, n^\beta]$, for $\beta > 0$. We say that a *labeled collision* occurs when there are two balls with the same labels in the same bin. We refer to this process as the β -process.

Lemma 4 *For each $a > 0$ there exists $\beta > 0$ and $c > 0$ such that when n balls are labeled with random integers in $[1, n^\beta]$ and next are thrown into $\frac{n}{\ln n}$ bins during the β -process then there are at most $c \ln n$ balls in every bin and no labeled collision occurs with probability $1 - n^{-a}$.*

Proof: We estimate from above the probabilities of the events that there are more than $c \ln n$ balls in some bin and that there is a labeled collision. We show that each of them can be made to be at most $n^{-a}/2$, from which it follows that some of these two events occurs with probability at most n^{-a} .

Let p denote the probability of selecting a specific bin when throwing a ball, which is $p = \frac{\ln n}{n}$. When we set $r = c \ln n$, for a sufficiently large $c > 1$, then

$$b(r; n, p) = \binom{n}{c \ln n} \left(\frac{\ln n}{n} \right)^{c \ln n} \left(1 - \frac{\ln n}{n} \right)^{n - c \ln n}. \quad (3)$$

Formula (3) translates (2) into the following bound

$$\Pr(S_n \geq r) \leq \binom{n}{c \ln n} \left(\frac{\ln n}{n} \right)^{c \ln n} \left(1 - \frac{\ln n}{n} \right)^{n - c \ln n} \cdot \frac{c \ln n (1 - \frac{\ln n}{n})}{c \ln n - \ln n}. \quad (4)$$

The right-hand side of (4) can be estimated by the following upper bound:

$$\begin{aligned} & \left(\frac{en}{c \ln n} \right)^{c \ln n} \left(\frac{\ln n}{n} \right)^{c \ln n} \left(1 - \frac{\ln n}{n} \right)^{n - c \ln n} \cdot \frac{c}{c-1} \\ &= \left(\frac{e}{c} \right)^{c \ln n} \left(1 - \frac{\ln n}{n} \right)^n \left(\frac{n}{n - \ln n} \right)^{c \ln n} \cdot \frac{c}{c-1} \\ &\leq n^c c^{-c \ln n} e^{-\ln n} \left(\frac{n}{n - \ln n} \right)^{c \ln n} \cdot \frac{c}{c-1} \\ &\leq n^{-c \ln c + c - 1}, \end{aligned}$$

for each sufficiently large $n > 0$. This is because

$$\left(\frac{n}{n - \ln n} \right)^{c \ln n} = \left(1 + \frac{\ln n}{n - \ln n} \right)^{c \ln n} \leq \exp \left(\frac{c \ln^2 n}{n - \ln n} \right),$$

which converges to 1. The probability that the number of balls in some bin is greater than $c \ln n$ is therefore at most $n \cdot n^{-c \ln c + c - 1} = n^{-c(\ln c - 1)}$, by the union bound. This probability can be made smaller than $n^{-a}/2$ for a sufficiently large $c > e$.

The probability of a labeled collision is at most that of a collision when n balls are thrown into n^β bins. This probability is at most $n^{-\beta+2}$ by bound (1) used in the proof of Lemma 3. This number can be made at most $n^{-a}/2$ for a sufficiently large β . \square

Next we summarize the performance of algorithm ARBITRARY-UNBOUNDED-LV as a Las Vegas algorithm.

Theorem 5 *Algorithm ARBITRARY-UNBOUNDED-LV terminates almost surely and there is no error when the algorithm terminates. For any $a > 0$, there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns names within $c \ln n$ time and generates at most $cn \ln n$ random bits with probability at least $1 - n^{-a}$.*

Proof: The algorithm terminates only when n different names have been assigned, which is provided by the condition that controls the main repeat-loop in Figure 3. This means that there is no error when the algorithm terminates.

We map executions of the β -process on executions of algorithm ARBITRARY-UNBOUNDED-LV in a natural manner. The main repeat-loop ends after an iteration in which each group of processors that select the same value for the variable `bin`, next select distinct values for the variable `label`. We interpret the random selections in an execution as throwing n balls into $\frac{n}{\ln n}$ bins, where a number `bin` determines a bin. The number of iterations of the inner repeat-loop equals the maximum number of balls in a bin.

For any $a > 0$, it follows that one iteration of the main repeat-loop suffices with probability at least $1 - n^{-a}$, for a suitable $\beta > 0$, by Lemma 4. It follows that i iterations are executed by termination with probability at most n^{-ia} , so the algorithm terminates almost surely.

Let us take $c > 0$ as in Lemma 4. It follows that an iteration of the main repeat-loop takes at most $c \ln n$ steps and one processor uses at most $c \ln n$ random bits in this one iteration with probability at least $1 - n^{-a}$. \square

Algorithm ARBITRARY-UNBOUNDED-LV is optimal among Las Vegas naming algorithms with respect to the following performance measures: the expected time $\mathcal{O}(\log n)$, by Theorem 3, the number of shared memory cells $\mathcal{O}(n/\log n)$ used to achieve this running time, by Corollary 1, both in Section 3, and the expected number of used random bits $\mathcal{O}(n \log n)$, by Proposition 1 in Section 2.

6 Las Vegas for Common with Bounded Memory

We consider the case of Common PRAM when the number of processors n is known and the number of available shared memory cells is constant. We propose a Las Vegas algorithm called COMMON-BOUNDED-LV, whose pseudocode is given in Figure 4.

An execution of the algorithm is organized as repeated “attempts” to assign temporary names. During such an attempt, each processor without a name chooses uniformly at random an integer in the interval $[1, \text{number-of-bins}]$, where `number-of-bins` is a parameter initialized to n ; such a selection is interpreted in a probabilistic analysis as throwing a ball into `number-of-bins` many bins. Next, for each $i \in [1, \text{number-of-bins}]$, the processors that selected i , if any, verify if they are unique in their selection of i by executing procedure VERIFY-COLLISION (given in Figure 1 in Section 2) $\beta \ln n$ times, where $\beta > 0$ is a number that is determined in analysis.

After no collision has been detected, a processor that selected i assigns itself a consecutive name by reading and incrementing the shared variable `Last-Name`. It takes up to $\beta \text{number-of-bins} \ln n$ verifications for collisions for all integers in $[1, \text{number-of-bins}]$. When this is over, the value of

Algorithm COMMON-BOUNDED-LV

```
repeat
  initialize number-of-bins  $\leftarrow n$  ; namev  $\leftarrow$  Last-Name  $\leftarrow 0$  ;
  no-collisionv  $\leftarrow$  true
  repeat
    initialize Collision-Detected  $\leftarrow$  false
    if namev = 0 then
      binv  $\leftarrow$  random integer in [1, number-of-bins]
      for  $i \leftarrow 1$  to number-of-bins do
        for  $j \leftarrow 1$  to  $\beta \ln n$  do
          if binv =  $i$  then
            if VERIFY-COLLISION then
              Collision-Detected  $\leftarrow$  collisionv  $\leftarrow$  true
            if binv =  $i$  and not collisionv then
              Last-Name  $\leftarrow$  Last-Name + 1
              namev  $\leftarrow$  Last-Name
        if  $n - \text{Last-Name} > \beta \ln n$ 
          then number-of-bins  $\leftarrow (n - \text{Last-Name})$ 
          else number-of-bins  $\leftarrow n/(\beta \ln n)$ 
      until not Collision-Detected
  until Last-Name =  $n$ 
```

Figure 4: A pseudocode for a processor v of a Common PRAM, where there is a constant number of shared memory cells. Procedure VERIFY-COLLISION has its pseudocode in Figure 1; lack of parameter means the default parameter 1. The variables Collision-Detected and Last-Name are shared. The private variable name stores the acquired name. The constant β is a parameter to be determined by analysis.

variable **number-of-bins** is modified by decrementing it by the number of new names just assigned, when working with the last **number-of-bins**, unless such decrementing would result in a number in **number-of-bins** that is at most $\beta \ln n$, in which case the variable **number-of-bins** is set to $n/(\beta \ln n)$. An attempt ends when all processors have tentative names assigned.

These names become final when there are a total of n of them, otherwise there are duplicates, so another attempt is performed. The main repeat loop in the pseudocode in Figure 4 represents an attempt to assign tentative names to each processor. An iteration of the inner repeat loop during which **number-of-bins** $> n/(\beta \ln n)$ is called *shrinking* and otherwise it is called *restored*.

Balls into bins. As a preparation for the analysis of performance of algorithm COMMON-BOUNDED-LV, we consider a related process of repeatedly throwing balls into bins, which we call the β -process. The β -process proceeds through *stages*, each representing one iteration of the

inner repeat-loop in Figure 4. A stage results in some balls removed and some transitioning to the next stage, so that eventually no balls remain and the process terminates.

The balls that participate in a stage are called *eligible* for the stage. In the first stage, n balls are eligible and we throw n balls into n bins. Initially, we apply the principle that after all eligible balls have been placed into bins during a stage, the singleton bins along with the balls in them are removed. A stage after which bins are removed is called *shrinking*. There are k bins and k balls in a shrinking stage; we refer to k as the *length* of this stage. Given balls and bins for any stage, we choose a bin uniformly at random and independently for each ball in the beginning of a stage and next place the balls in their selected destinations. The bins that either are empty or multiple in a shrinking stage stay for the next stage. The balls from multiple bins become eligible for the next stage.

This continues until such a shrinking stage after which at most $\beta \ln n$ balls remain. Then we restore bins for a total of $n/(\beta \ln n)$ of them to be used in the following stages, during which we never remove any bin; these stages are called *restored*. In these final restored stages, we keep removing singleton balls at the end of a stage, while balls from multiple bins stay as eligible for the next restored stage. This continues until all balls are removed.

Lemma 5 *For any $a > 0$, there exists $\beta > 0$ such that the sum of lengths of all shrinking stages in the β -process is at most $2en$, where e is the base of natural logarithms, and there are at most $\beta \ln n$ restored stages, both events holding with probability $1 - n^{-a}$, for sufficiently large n .*

Proof: We consider two cases depending on the kind of analyzed stages. Let $k \leq n$ denote the length of a stage.

In a shrinking stage, we throw k balls into k bins, while choosing bins independently and uniformly at random. The probability that a ball ends up singleton can be bounded from below as follows:

$$\begin{aligned} k \cdot \frac{1}{k} \left(1 - \frac{1}{k}\right)^{k-1} &\geq \left(e^{-\frac{1}{k} - \frac{1}{k^2}}\right)^{k-1} \\ &= e^{-\frac{k-1}{k} - \frac{k-1}{k^2}} \\ &= e^{-1 + \frac{1}{k} - \frac{1}{k} + \frac{1}{k^2}} \\ &= \frac{1}{e} \cdot e^{1/k^2} \\ &\geq \frac{1}{e}, \end{aligned}$$

where we used the inequality $1 - x \geq e^{-x-x^2}$, which holds for $0 \leq x \leq \frac{1}{2}$.

Let Z_k be the number of singleton balls after k balls are thrown into k bins. It follows that the expectancy of Z_k satisfies $\mathbb{E}[Z_k] \geq k/e$.

To estimate the deviation of Z_k from its expected value, we use the bounded differences inequality [49, 52]. Let B_j be the bin of ball b_j , for $1 \leq j \leq k$. Then Z_k is of the form $Z_k = h(B_1, \dots, B_k)$ where h satisfied the Lipschitz condition with constant 2, because moving one ball to a different bin results in changing the value of h by at most 2 with respect to the original value. The bounded-differences inequality specialized to this instance is as follows, for any $d > 0$:

$$\Pr(Z_k \leq \mathbb{E}[Z_k] - d\sqrt{k}) \leq \exp(-d^2/8). \quad (5)$$

We use this inequality for $d = \frac{\sqrt{k}}{2e}$. Then (5) implies the following bound:

$$\begin{aligned} \Pr\left(Z_k \leq \frac{k}{e} - \frac{k}{2e}\right) &= \Pr\left(Z_k \leq \frac{k}{2e}\right) \\ &\leq \exp\left(-\frac{1}{8} \cdot \left(\frac{\sqrt{k}}{2e}\right)^2\right) \\ &= \exp\left(-\frac{k}{32e^2}\right). \end{aligned}$$

If we start a shrinking stage with k eligible balls then the number of balls eligible for the next stage is at most

$$\left(1 - \frac{1}{2e}\right) \cdot k = \frac{2e-1}{2e} \cdot k,$$

with probability at least $1 - \exp(-k/32e^2)$. Let us continue shrinking stages as long as the inequality

$$\frac{k}{32e^2} > 3a \ln n$$

holds. We denote this inequality concisely as $k > \beta \ln n$ for $\beta = 96e^2a$. Then the probability that every shrinking stage results in the size of the pool of eligible balls decreasing by a factor of at least

$$\frac{2e-1}{2e} = \frac{1}{f}$$

is itself at least

$$\left(1 - e^{-3a \ln n}\right)^{\log_f n} \geq 1 - \frac{\log_f n}{n^{-3a}} \geq 1 - n^{-2a},$$

for sufficiently large n , by Bernoulli's inequality.

If all shrinking stages result in the size of the pool of eligible balls decreasing by a factor of at least $1/f$, then the total number of eligible balls summed over all such stages is at most

$$n \sum_{i \geq 0} f^{-i} = n \cdot \frac{1}{1-f^{-1}} = 2en.$$

In a restored stage, there are at most $\beta \ln n$ eligible balls. A restored stage happens to be the last one when all the balls become single after their placement, which occurs with probability at least

$$\left(\frac{n/(\beta \ln n) - \beta \ln n}{n/(\beta \ln n)}\right)^{\beta \ln n} = \left(1 - \frac{\beta^2 \ln^2 n}{n}\right)^{\beta \ln n} \geq 1 - \frac{\beta^3 \ln^3 n}{n},$$

by the Bernoulli's inequality. It follows that there are more than $\beta \ln n$ restored stages with probability at most

$$\left(\frac{\beta^3 \ln^3 n}{n}\right)^{\beta \ln n} = n^{-\Omega(\log n)}.$$

This bound is at most n^{-2a} for sufficiently large n .

Both events, one about shrinking stages and the other about restored stages, hold with probability at least $1 - 2n^{-2a} \geq 1 - n^{-a}$, for sufficiently large n . \square

Next we summarize the performance of algorithm COMMON-BOUNDED-LV as Las Vegas one. In its proof, we rely on mapping executions of the β -process on executions of algorithm COMMON-BOUNDED-LV in a natural manner.

Theorem 6 *Algorithm COMMON-BOUNDED-LV terminates almost surely and there is no error when the algorithm terminates. For any $a > 0$ there exist $\beta > 0$ and $c > 0$ such that the algorithm terminates within time $cn \ln n$ using at most $cn \ln n$ random bits with probability $1 - n^{-a}$.*

Proof: The condition controlling the main repeat-loop guarantees that an execution terminates only when the assigned names fill the interval $[1, n]$, so they are distinct and there is no error.

To analyze time performance, we consider the β -process of throwing balls into bins as considered in Lemma 5. Let $\beta_1 > 0$ be the number β specified in this Lemma, as determined by a replaced by $2a$ in its assumptions. This Lemma gives that the sum of all values of K summed over all shrinking stages is at most $2en$ with probability at least $1 - n^{-2a}$.

For a given K and a number $i \in [1, K]$, procedure VERIFY-COLLISION is executed $\beta \ln n$ times, where β is the parameter in Figure 4. If there is a collision then it is detected with probability at least $2^{-\beta \ln n}$. We may take $\beta_2 \geq \beta_1$ sufficiently large so that the inequality $2en \cdot 2^{-\beta_2 \ln n} < n^{-2a}$ holds.

The total number of instances of executing VERIFY-COLLISION during an iteration of the main loop, while K is kept equal to $n/(\beta \ln n)$, is at most n . Observe that the inequality $n \cdot 2^{-\beta_2 \ln n} < n^{-2a}$ holds with probability at most $1 - n^{-2a}$ because $n < 2en$.

If β is set in Figure 4 to β_2 then one iteration of the outer repeat-loop suffices with probability at least $1 - 2n^{-2a}$, for sufficiently large n . This is because verifications for collisions detect all existing collisions with this probability. Similarly, this one iteration takes $\mathcal{O}(n \log n)$ time with probability that is at least $1 - 2n^{-2a}$, for sufficiently large n . The claimed performance holds therefore with probability at least $1 - n^{-a}$, for sufficiently large n .

There are at least i iterations of the main repeat-loop with probability at most n^{-ia} , so the algorithm terminates almost surely. \square

Algorithm COMMON-BOUNDED-LV is optimal among Las Vegas algorithms with respect to the following performance measures: the expected time $\mathcal{O}(n \log n)$, given the amount $\mathcal{O}(1)$ of its available shared memory, by Corollary 1 in Section 3, and the expected number of random bits $\mathcal{O}(n \log n)$, by Proposition 1 in Section 2.

7 Las Vegas for Common with Unbounded Memory

We consider now the last case when the number of processors n is known. The PRAM is of its Common variant, and there is an unbounded amount of shared memory. We propose a Las Vegas algorithm called COMMON-UNBOUNDED-LV, the pseudocode for this algorithm is given in Figure 5. Subroutines of prefix-type, like computing the number of selects and ranks of selected numbers are not included in this pseudocode. The algorithm invokes procedure VERIFY-COLLISION, whose pseudocode is in Figure 1.

An execution of algorithm COMMON-UNBOUNDED-LV proceeds as a sequence of attempts to assign temporary names. When such an attempt results in assigning temporary names without duplicates then these transient names become final. An attempt begins from each processor selecting an integer from the interval $[1, (\beta + 1)n]$ uniformly at random and independently, where β is a parameter such that only $\beta > 1$ is assumed. Next, for $\lg n$ steps, each process executes procedure VERIFY-COLLISION(x) where x is the currently selected integer. If a collision is detected then a

Algorithm COMMON-UNBOUNDED-LV

```
binv ← random integer in [1, (β + 1)n]          /* throw a ball into binv */
repeat
  for i ← 1 to lg n do
    if VERIFY-COLLISION(binv) then
      binv ← random integer in [1, (β + 1)n]
    number-occupied-bins ← total number of currently selected values for binv
until number-occupied-bins = n
namev ← the rank of binv among nonempty bins
```

Figure 5: A pseudocode for a processor v of a Common PRAM, where the number of shared memory cells is unbounded. The constant β is a parameter that satisfies the inequality $\beta > 1$. The private variable **name** stores the acquired name.

processor immediately selects another number in $[1, (\beta + 1)n]$ and continues verifying for a collision. After $\lg n$ such steps, the processors count the total number of selections of different integers. If this number equals exactly n then the ranks of the selected integers are assigned as names, otherwise another attempt to find names is made. Computing the number of selections and the ranks takes time $\mathcal{O}(\log n)$. In order to amortize this time $\mathcal{O}(\log n)$ by verifications, such a computation of ranks is performed only after $\lg n$ verifications. Here a rank of a selected x is the number of selected numbers that are at most x .

Balls into bins. We consider auxiliary processes of placing balls into bins that abstracts operations on shared memory as performed by algorithm COMMON-UNBOUNDED-LV.

The β -process is about placing n balls into $(\beta + 1)n$ bins. The process is structured as a sequence of stages. A stage represents an abstraction of one iteration of the inner for-loop in Figure 5 performed as if collisions were detected instantaneously and with certainty. When a ball is moved then it is placed in a bin selected uniformly at random, all such selections independent from one another. The stages are performed as follows. In the first stage, n balls are placed into $(\beta + 1)n$ bins. When a bin is singleton in the beginning of a stage then the ball in the bin stays put through the stage. When a bin is multiple in the beginning of a stage, then all the balls in this bin participate actively in this stage: they are removed from the bin and placed in randomly-selected bins. The process terminates after a stage in which all balls reside in singleton bins.

In analysis, it is convenient to visualize a stage as occurring by first removing all balls from multiple bins and then placing the removed balls in randomly selected bins one by one. We model placements of single balls by movements of a random walk. More precisely, we associate a *mimicking walk* to each execution of the β -process. Such a walk is performed on points with integer coordinates on a line, as explained next in detail.

The mimicking walk proceeds through stages, similarly as the ball process. When we are to relocate k balls in a stage of the ball process then this is represented by the mimicking walk

starting the corresponding stage at coordinate k . Suppose that we process a ball in a stage and the mimicking walk is at some position i . Placing this ball in an empty bin temporarily decreases the number of balls for the next stage; the respective action in the mimicking walk is to decrement its position from i to $i - 1$. Placing this ball in an occupied bin temporarily increases the number of balls for the next stage; the respective action in the mimicking walk is to increment its position from i to $i + 1$. The mimicking walk gives a conservative estimates on the behavior of the ball process, as we show next.

Lemma 6 *If a stage of the mimicking walk ends at a position k , then the corresponding stage of the ball β -process ends with at most k balls to be relocated into bins in the next stage.*

Proof: The argument is broken into three cases, in which we consider what happens in the ball β -process and what are the corresponding actions in the mimicking walk. A number of balls in a bin in a stage is meant to be the final number of balls in this bin at the end of the stage.

In the first case, just one ball is placed in a bin that begins the stage as empty. Then this ball will not be relocated in the next stage. This means that the number of balls for the next stage decreases by 1. At the same time, the mimicking walk decrements its position by 1.

In the second case, some $j \geq 1$ balls land in a bin that is singleton at the start of this stage, so this ball was not eligible for the stage. Then the number of balls in the bin becomes $j + 1$ and these many balls will need to be relocated in the next stage. Observe that this contributes to incrementing the number of the eligible balls in the next stage by 1, because only the original ball residing in the singleton bin is added to the set of eligible balls, while the other balls participate in both stages. At the same time, the mimicking walk increments its position j times, by 1 each time.

In the third and final case, some $j \geq 2$ balls land in a bin that is empty at the start of this stage. Then this does not contribute to a change in the number of balls eligible for relocation in the next stage, as these j balls participate in both stages. Let us consider these balls as placed in the bin one by one. The first ball makes the mimicking walk decrement its position, as the ball is single in the bin. The second ball makes the walk increment its position, so that it returns to the original position as at the start of the stage. The following ball placements, if any, result in the walk incrementing its positions. \square

Random walks. Next we consider a random walk which will estimate the behavior of a ball process. One component of estimation is provided by Lemma 6, in that we will interpret a random walk as a mimicking walk for the ball process.

The random walk is represented as movements of a marker placed on the non-negative side of the integer number line. The movements of the marker are by distance 1 at a time and they are independent from each other. The *random β -walk* has the marker's position incremented with probability $\frac{1}{\beta+1}$ and decremented with probability $\frac{\beta}{\beta+1}$. This may be interpreted as a sequence of independent Bernoulli trials, in which $\frac{\beta}{\beta+1}$ is chosen to be the probability of success. We will consider $\beta > 1$, for which $\frac{\beta}{\beta+1} > \frac{1}{\beta+1}$, which means that the probability of success is greater than the probability of failure.

Such a random β -walk proceeds through *stages*, which are defined as follows. The first stage begins at position n . When a stage begins at a position k then it ends after k moves, unless it reaches the zero coordinate in the meantime. The zero point acts as an absorbing barrier, and when

the walk's position reaches it then the random walk terminates. This is the only way in which the walk terminates. A stage captures one round of PRAM's computation and the number of moves in a stage represents the number of writes processors perform in a round.

Lemma 7 *For any numbers $a > 0$ and $\beta > 1$, there exists $b > 0$ such that the random β -walk starting at position $n > 0$ terminates within $b \ln n$ stages with all of them comprising $\mathcal{O}(n)$ moves with probability at least $1 - n^{-a}$.*

Proof: Suppose the random walk starts at position $k > 0$ when a stage begins. Let X_k be the number of moves towards 0 and $Y_k = k - X_k$ be the number of moves away from 0 in such a stage. The total distance covered towards 0, which we call a *drift*, is

$$L(k) = X_k - Y_k = X_k - (k - X_k) = 2X_k - k .$$

The expected value of X_k is $\mathbb{E}[X_k] = \frac{\beta k}{\beta+1} = \mu_k$. The event $X_k < (1-\varepsilon)\mu_k$ holds with probability at most $\exp(-\frac{\varepsilon^2}{2}\mu_k)$, by the Chernoff bound [52], so that $X_k \geq (1-\varepsilon)\mu_k$ occurs with probability at least $1 - \exp(-\frac{\varepsilon^2}{2}\mu_k)$. We say that such a stage is *conforming* when the event $X_k \geq (1-\varepsilon)\mu_k$ holds.

If a stage is conforming then the following inequality holds:

$$L(k) \geq 2(1-\varepsilon)\frac{\beta k}{\beta+1} - k = \frac{\beta - 2\beta\varepsilon - 1}{\beta+1}k .$$

We want the inequality $\frac{\beta - 2\beta\varepsilon - 1}{\beta+1} > 0$ to hold, which is the case when $\varepsilon < \frac{\beta-1}{2\beta}$. Let us fix such $\varepsilon > 0$. Now the distance from 0 after k steps starting at k is

$$k - L(k) = (1 - \frac{\beta - 2\beta\varepsilon - 1}{\beta+1}) \cdot k = \frac{2(1+\beta\varepsilon)}{\beta+1} \cdot k ,$$

where $\frac{2(1+\beta\varepsilon)}{\beta+1} < 1$ for $\varepsilon < \frac{\beta-1}{2\beta}$. Let $\rho = \frac{\beta+1}{2(1+\beta\varepsilon)} > 1$. Consecutive i conforming stages make the distance from 0 decrease by at least a factor ρ^{-i} .

When we start the first stage at position n and the next $\log_\rho n$ stages are conforming then after these many stages the random walk ends up at a position that is close to 0. For our purposes, it suffices that the position is of distance at most $s \ln n$ from 0, for some $s > 0$, because of its impact on the probability estimates. Namely, the event that all these stages are conforming and the bound $s \ln n$ on distance from 0 holds, occurs with probability at least

$$1 - \log_\rho n \cdot \exp(-\frac{\varepsilon^2}{2} \frac{\beta}{\beta+1} s \ln n) \geq 1 - \log_\rho n \cdot n^{-\frac{\varepsilon^2}{2} \frac{\beta}{\beta+1} s} .$$

Let us choose $s > 0$ such that

$$\log_\rho n \cdot n^{-\frac{\varepsilon^2}{2} \frac{\beta}{\beta+1} s} \leq \frac{1}{2n^a} ,$$

for sufficiently large n .

Having fixed s , let us take $t > 0$ such that the distance covered towards 0 is at least $s \ln n$ when starting from $k = t \ln n$ and performing k steps. We interpret these movements as if this was a single conceptual stage for the sake of the argument, but its duration comprises all stages when we

start from $s \ln n$ until we terminate at 0. It follows that the conceptual stage comprises at most $t \ln n$ real stages, because a stage takes at least one round.

If this last conceptual stage is conforming then the distance covered towards 0 is bounded by

$$L(k) \geq \frac{\beta - 2\beta\varepsilon - 1}{\beta + 1} \cdot k .$$

We want this to be at least $s \ln n$ for $k = t \ln n$, which is equivalent to

$$\frac{\beta - 2\beta\varepsilon - 1}{\beta + 1} \cdot t > s .$$

Now it is sufficient to take $t > s \cdot \frac{\beta+1}{\beta-2\beta\varepsilon-1}$. This last conceptual stage is not conforming with probability at most $\exp(-\frac{\varepsilon^2}{2} \frac{\beta}{\beta+1} t \ln n)$. Let us take t that is additionally big enough for the following inequality

$$\exp(-\frac{\varepsilon^2}{2} \frac{\beta}{\beta+1} t \ln n) = n^{-\frac{\varepsilon^2}{2} \frac{\beta}{\beta+1} t} \leq \frac{1}{2n^a}$$

to hold.

Having selected s and t , we can conclude that there are at most $(s+t) \ln n$ stages with probability at least $1 - n^{-a}$.

Now, let us consider only the total number of moves to the left X_m and to the right Y_m after m moves in total, when starting at position n . The event $X_m < (1-\varepsilon) \cdot \frac{\beta}{1+\beta} \cdot m$ holds with probability at most $\exp(-\frac{\varepsilon^2}{2} \frac{\beta}{1+\beta} \cdot m)$, by the Chernoff bound [52], so that $X_m \geq m \cdot \frac{(1-\varepsilon)\beta}{1+\beta}$ occurs with at least the respective high probability $1 - \exp(-\frac{\varepsilon^2}{2} \frac{\beta}{1+\beta} \cdot m)$. At the same time, we have that the number of moves away from zero, which we denote Y_m , can be estimated to be

$$Y_m = m - X_m < m - m \cdot \frac{(1-\varepsilon)\beta}{1+\beta} = \frac{1+\varepsilon\beta}{1+\beta} \cdot m .$$

This gives an estimate on the corresponding drift:

$$L(m) = X_m - Y_m > \frac{\beta - 2\beta\varepsilon - 1}{\beta + 1} \cdot m .$$

We want the inequality $\frac{\beta-2\beta\varepsilon-1}{\beta+1} > 0$ to hold, which is the case when $\varepsilon < \frac{\beta-1}{2\beta}$. The drift is at least n , with the corresponding large probability, when $m = d \cdot n$ for $d = \frac{\beta+1}{\beta-2\beta\varepsilon-1}$. The drift is at least such with probability exponentially close to 1 in n , which is at least $1 - n^{-a}$ for sufficiently large n . \square

Lemma 8 *For any numbers $a > 0$ and $\beta > 1$, there exists $b > 0$ such that the β -process starting at position $n > 0$ terminates within $b \ln n$ stages after performing $\mathcal{O}(n)$ ball throws with probability at least $1 - n^{-a}$.*

Proof: We estimate the behavior of the β -process on n balls by the behavior of the random β -walk starting at position n . The justification of the estimation is in two steps. One is the property of mimicking walks given as Lemma 6. The other is provided by Lemma 7 and is justified as follows. The probability of decrementing and incrementing position in the random β -walk are such that

they reflect the probabilities of landing in an empty bin or in an occupied bin. Namely, we use the facts that during executing the β -process, there are at most n occupied bins and at least $\beta \cdot n$ empty bins in any round. In the β -process, the probability of landing in an empty bin is at least $\frac{\beta n}{(\beta+1)n} = \frac{\beta}{\beta+1}$, and the probability of landing in an occupied bin is at most $\frac{n}{(\beta+1)n} = \frac{1}{\beta+1}$. This means that the random β -walk is consistent with Lemma 6 in providing estimates on the time of termination of the β -process from above. \square

Incorporating verifications. We consider the *random β -walk with verifications*, which is defined as follows. The process proceeds through stages, similarly as the regular random β -walk. For any round of the walk and a position at which the walk is at, we first perform a Bernoulli trial with the probability $\frac{1}{2}$ of success. Such a trial is referred to as a *verification*, which is *positive* when a success occurs, otherwise it is *negative*. A positive verification results in a movement of the marker as in the regular β -walk, otherwise the walk pauses at the given position for this round.

Lemma 9 *For any numbers $a > 0$ and $\beta > 1$, there exists $b > 0$ such that the random β -walk with verifications starting at position $n > 0$ terminates within $b \ln n$ stages with all of them comprising the total of $\mathcal{O}(n)$ moves with probability at least $1 - n^{-a}$.*

Proof: We provide an extension of the proof of Lemma 7, which states a similar property of regular random β -walks. That proof estimated times of stages and the number of moves. Suppose the regular random β -walk starts at a position k , so that the stage takes k moves. There is a constant $d < 1$ such that the walk ends at a position at most dk with probability exponential in k .

Moreover, the proof of Lemma 7 is such that all the values of k considered are at least logarithmic in n , which provides at most a polynomial bound on error. A random walk with verifications is slowed down by negative verifications. Observe that a random walk with verifications that is performed $3k$ times undergoes at least k positive verifications with probability exponential in k by the Chernoff bound [52]. This means that the proof of Lemma 7 can be adapted to the case of random walks with verifications almost verbatim, with the modifications contributed by polynomial bounds on error of estimates of the number of positive verifications in stages. \square

Next, we consider a *β -process with verifications*, which is defined as follows. The process proceeds through stages, similarly as the regular ball process. The first stage starts with placing n balls into $(\beta + 1)n$ bins. For any following stage, we first go through multiple bins and, for each ball in such a bin, we perform a Bernoulli trial with the probability $\frac{1}{2}$ of success, which we call a *verification*. A success in a trial is referred to as a *positive verification* otherwise it is a *negative* one. If at least one positive verification occurs for a ball in a multiple bin then all the balls in this bin are relocated in this stage to bins selected uniformly at random and independently for each such a ball, otherwise the balls stay put in this bin until the next stage. The β -process terminates when all the balls are singleton.

Lemma 10 *For any numbers $a > 0$ and $\beta > 1$, there exists $b > 0$ such that the β -process with verifications terminates within $b \ln n$ stages with all of them comprising the total of $\mathcal{O}(n)$ ball throws with probability at least $1 - n^{-a}$.*

Proof: The argument proceeds by combining Lemma 6 with Lemma 9, similarly as the proof of Lemma 8 is proved by combining Lemma 6 with Lemma 7. The details follow.

For any execution of a ball process with verifications, we consider a “mimicking random walk,” also with verifications, defined such that when a ball from a multiple bin is handled then the outcome of a random verification for this ball is mapped on a verification for the corresponding random walk. Observe that for a β -process with verifications just one positive verification is sufficient among $j - 1$ trials when there are $j > 1$ balls in a multiple bin, so a random β -walk with verifications provides an upper bound on time of termination of the β -process with verifications. The probabilities of decrementing and incrementing position in the random β -walk with verifications are such that they reflect the probabilities of landing in an empty bin or in an occupied bin, similarly as without verifications. All this give a consistency of a β -walk with verifications with Lemma 6 in providing estimates on the time of termination of the β -process from above. \square

Next we summarize the performance of algorithm COMMON-UNBOUNDED-LV as Las Vegas one. The proof is based on mapping executions of the β -processes with verifications on executions of algorithm COMMON-UNBOUNDED-LV in a natural manner.

Theorem 7 *Algorithm COMMON-UNBOUNDED-LV terminates almost surely and when the algorithm terminates then there is no error. For each $a > 0$ and any $\beta > 1$ in the pseudocode, there exists $c > 0$ such that the algorithm assigns proper names within time $c \lg n$ and using at most $c n \lg n$ random bits with probability at least $1 - n^{-a}$.*

Proof: The algorithm terminates when there are n different ranks, by the condition controlling the repeat-loop. As ranks are distinct and each in the interval $[1, n]$, each name is unique, so there is no error. The repeat-loop is executed $\mathcal{O}(1)$ times with probability at least $1 - n^{-a}$, by Lemma 10. The repeat-loop is performed i times with probability that is at most n^{-ia} , so it converges to 0 with i increasing. It follows that the algorithm terminates almost surely.

An iteration of the repeat-loop in Figure 5 takes $\mathcal{O}(\log n)$ steps. This is because of the following two facts. First, it consists of $\lg n$ iterations of the for-loop, each taking $\mathcal{O}(1)$ rounds. Second, it concludes with verifying the until-condition, which is carried out by counting nonempty bins by a prefix-type computation. It follows that time until termination is $\mathcal{O}(\log n)$ with probability $1 - n^{-a}$.

By Lemma 10, the total number of ball throws is $\mathcal{O}(n)$ with probability $1 - n^{-a}$. Each placement of a ball requires $\mathcal{O}(\log n)$ random bits, so the number of used random bits is $\mathcal{O}(n \log n)$ with the same probability. \square

Algorithm COMMON-UNBOUNDED-LV is optimal among Las Vegas naming algorithms with respect to the following performance measures: the expected time $\mathcal{O}(\log n)$, by Theorem 3, the number of shared memory cells $\mathcal{O}(n)$ used to achieve this running time, by Corollary 1, and the expected number of random bits $\mathcal{O}(n \log n)$, by Proposition 1.

8 Monte Carlo for Arbitrary with Bounded Memory

We develop a Monte Carlo naming algorithm for an Arbitrary PRAM with a constant number of shared memory cells, when the number of processors n is unknown. The algorithm is called ARBITRARY-BOUNDED-MC and its pseudocode is given in Figure 6.

The underlying idea is to have all processors repeatedly attempt to obtain tentative names and terminate when the probability of duplicate names is gauged to be sufficiently small. To this end,

Algorithm ARBITRARY-BOUNDED-MC

```
initialize  $k \leftarrow 1$                                 /* initial approximation of  $\lg n$  */
repeat
  initialize Last-Name  $\leftarrow$  name $v$   $\leftarrow$  0
   $k \leftarrow 2k$ 
  bin $v$   $\leftarrow$  random integer in  $[1, 2^k]$            /* throw a ball into a bin */
  repeat
    All-Named  $\leftarrow$  true
    if name $v$  = 0 then
      Pad  $\leftarrow$  bin $v$ 
      if Pad = bin $v$  then
        Last-Name  $\leftarrow$  Last-Name + 1
        name $v$   $\leftarrow$  Last-Name
      else
        All-Named  $\leftarrow$  false
  until All-Named
until Last-Name  $\leq 2^{k/\beta}$ 
```

Figure 6: A pseudocode for a processor v of an Arbitrary PRAM with a constant number of shared memory cells. The variables **Last-Name**, **All-Named** and **Pad** are shared. The private variable **name** stores the acquired name. The constant $\beta > 0$ is a parameter to be determined by analysis.

each processor writes an integer selected from a suitable “selection range” into a shared memory register and next reads this register to verify whether the write was successful or not. A successful write results in each such a processor getting a tentative name by reading and incrementing another shared register operating as a counter. One of the challenges here is to determine a selection range from which random integers are chosen for writing. A good selection range is large enough with respect to the number of writers, which is unknown, because when the range is too small then multiple processors may select the same integer and so all of them get the same tentative name after this integer gets written successfully. The algorithm keeps the size of a selection range growing with each failed attempt to assign tentative names.

There is an inherent tradeoff here, since on the one hand, we want to keep the size of used shared memory small, as a measure of efficiency of the algorithm, while, at the same time, the larger the range of memory the smaller the probability of collision of random selections from a selection range and so of the resulting duplicate names. Additionally, increasing the selection range repeatedly costs time for each such a repetition, while we also want to minimize the running time as the metric of performance. The algorithm keeps increasing the selection range with a quadratic rate, which turns out to be sufficient to optimize all the performance metrics we measure. The

algorithm terminates when the number of selected integers from the current selection range makes a sufficiently small fraction of the size of the used range.

The structure of the pseudocode in Figure 6 is determined by the main repeat-loop. Each iteration of this loop begins with doubling the variable k , which determines the selection range $[1, 2^k]$. This means that the size of the selection range increases quadratically with consecutive iterations of the main repeat-loop. A processor begins an iteration of the main loop by choosing an integer uniformly at random from the current selection range $[1, 2^k]$. There is an inner repeat-loop, nested within the main loop, which assigns tentative names depending on the random selections just made.

All processors repeatedly write to a shared variable **Pad** and next read to verify if the write was successful. It is possible that different processors attempt to write the same value and then verify that their write was successful. The shared variable **Last-Name** is used to proceed through consecutive integers to provide tentative names to be assigned to the latest successful writers. When multiple processors attempt to write the same value to **Pad** and it gets written successfully, then all of them obtain the same tentative name. The variable **Last-Name**, at the end of each iteration of the inner repeat-loop, equals the number of occupied bins. The shared variable **All-Named** is used to verify if all processors have tentative names. The outer loop terminates when the number of assigned names, which is the same as the number of occupied bins, is smaller than or equal to $2^{k/\beta}$, where $\beta > 0$ is a parameter to be determined in analysis.

Balls into bins. We consider the following auxiliary β -process of throwing balls into bins, for a parameter $\beta > 0$. The process proceeds through stages identified by consecutive positive integers. The i th stage has the number parameter k equal to $k = 2^i$. During a stage, we first throw n balls into the corresponding 2^k bins and next count the number of occupied bins. A stage is last in an execution of the β -process, and so the β -process terminates, when the number of occupied bins is smaller than or equal to $2^{k/\beta}$.

We may observe that the β -process always terminates. This is because, by its specification, the β -process terminates by the first stage in which the inequality $n \leq 2^{k/\beta}$ holds, where n is an upper bound on the number of occupied bins in a stage. The inequality $n \leq 2^{k/\beta}$ is equivalent to $n^\beta \leq 2^k$ and so to $\beta \lg n \leq k$. Since k goes through consecutive powers of 2, we obtain that the number of stages of the β -process with n balls is at most $\lg(\beta \lg n) = \lg \beta + \lg \lg n$.

We say that such a β -process is *correct* when upon termination each ball is in a separate bin, otherwise the process is *incorrect*.

Lemma 11 *For any $a > 0$ there exists $\beta > 0$ such that the β -process is incorrect with probability that is at most n^{-a} , for sufficiently large n .*

Proof: The β -process is incorrect when there are collisions after the last stage. The probability of the intersection of the events “ β -process terminates” and “there are collisions” is bounded from above by the probability of each of these two events. Next we show that, for each pair of k and n , some of these two events occurs with probability that is at most n^{-a} , for a suitable β .

First, we consider the event that the β -process terminates. The probability that there are at

most $2^{k/\beta}$ occupied bins is at most

$$\begin{aligned}
\binom{2^k}{2^{k/\beta}} \left(\frac{2^{k/\beta}}{2^k} \right)^n &\leq \left(\frac{e 2^k}{2^{k/\beta}} \right)^{2^{k/\beta}} 2^{k(\beta^{-1}-1)n} \\
&\leq e^{2^{k/\beta}} \cdot 2^{k(1-\beta^{-1})2^{k/\beta}} \cdot 2^{k(\beta^{-1}-1)n} \\
&\leq e^{2^{k/\beta}} \cdot 2^{k(\beta^{-1}-1)(n-2^{k/\beta})} .
\end{aligned} \tag{6}$$

We estimate from above the natural logarithm of the right-hand side of (6). We obtain the following upper bound:

$$\begin{aligned}
2^{k/\beta} + k(\beta^{-1} - 1)(n - 2^{k/\beta}) \ln 2 &< 2^{k/\beta} - \frac{1}{2}(n - 2^{k/\beta}) \ln 2 \\
&= 2^{k/\beta} - \frac{\ln 2}{2}n + \frac{\ln 2}{2}2^{k/\beta} \\
&= -\frac{\ln 2}{2}n + 2^{k/\beta} \cdot \frac{2 + \ln 2}{2} ,
\end{aligned} \tag{7}$$

for $\beta > 4/3$, as $k \geq 2$. The estimate (7) is at most $-n \cdot \frac{\ln 2}{4}$ when $2^{k/\beta} \leq n \cdot \delta$, for $\delta = \frac{\ln 2}{2(2+\ln 2)}$, by a direct algebraic verification. These restrictions on k and β can be restated as

$$k \leq \beta \lg(n\delta) \text{ and } \beta > 4/3 . \tag{8}$$

When this condition (8) is satisfied, then the probability of at most $2^{k/\beta}$ occupied bins is at most

$$\exp\left(-n \cdot \frac{\ln 2}{4}\right) \leq n^{-a}$$

for sufficiently large n .

Next, let us consider the probability of collisions occurring. Collisions do not occur with probability that is at least

$$\left(1 - \frac{n}{2^k}\right)^n \geq 1 - \frac{n^2}{2^k} ,$$

by the Bernoulli's inequality. It follows that the probability of collisions occurring can be bounded from above by $\frac{n^2}{2^k}$. This bound in turn is at most n^{-a} when

$$k \geq (2 + a) \lg n . \tag{9}$$

In order to have some of the inequalities (8) and (9) hold for any k and n , it is sufficient to have

$$(2 + a) \lg n \leq \beta \lg(n\delta) .$$

This determines β as follows:

$$\beta \geq \frac{(2 + a) \lg n}{\lg n + \lg \delta} \rightarrow 2 + a ,$$

with $n \rightarrow \infty$. We obtain that the inequality $\beta > 2 + a$ suffices, for n that is large enough. \square

Lemma 12 *For each $\beta > 0$ there exists $c > 0$ such that when the β -process terminates then the number of bins ever needed is at most cn and the number of random bits ever generated is at most $cn \ln n$.*

Proof: The β -process terminates by the stage in which the inequality $n \leq 2^{k/\beta}$ holds, so k gets to be at most $\beta \lg n$. We partition the range $[2, \beta \lg n]$ of values of k into two subranges and consider them separately.

First, when k ranges from 2 to $\lg n$ through the stages, then the numbers of needed bins increase quadratically through the stages, because k is doubled with each transition to the next stage. This means that the total number of all these bins is $\mathcal{O}(n)$. At the same time, the number of random bits increases geometrically through the stages, so the total number of random bits a processor uses is $\mathcal{O}(\log n)$.

Second, when k ranges from $\lg n$ to $\beta \lg n$, the number of needed bins is at most n in each stage. There are only $\lg(\beta + 1)$ such stages, so the total number of all these bins is $\lg(\beta + 1) \cdot n$. At the same time, a processor uses at most $\beta \lg n$ random bits in each of these stages. \square

There is a direct correspondence between iterations of the outer repeat-loop and stages of a β -process. The i th stage has the number k equal to the value of k during the i th iteration of the outer repeat-loop of algorithm ARBITRARY-BOUNDED-MC, that is, we have $k = 2^i$. We map an execution of the algorithm into a corresponding execution of a β -process in order to apply Lemmas 11 and 12 in the proof of the Theorem 8, which summarizes the performance of algorithm ARBITRARY-BOUNDED-MC and justifies that it is Monte Carlo.

Theorem 8 *Algorithm ARBITRARY-BOUNDED-MC always terminates, for any $\beta > 0$. For each $a > 0$ there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns unique names, works in time at most cn , and uses at most $cn \ln n$ random bits, all this with probability at least $1 - n^{-a}$.*

Proof: The number of stages of the β -process with n balls is at most $\lg(\beta \lg n) = \lg \beta + \lg \lg n$. This is also an upper bound on the number of iterations of the main repeat-loop. We conclude that the algorithm always terminates.

The number of bins available in a stage is an upper bound on the number of bins occupied in this stage. The number of bins occupied in a stage equals the number of times the inner repeat-loop is iterated, because executing instruction `Pad \leftarrow bin` eliminates one occupied bin. It follows that the number of bins ever needed is an upper bound on time of the algorithm. The number of iterations of the inner repeat-loop is recorded in the variable `Last-Name`, so the termination condition of the algorithm corresponds to the termination condition of the β -process.

When the β -process is correct then this means that the processors obtain distinct names. We conclude that Lemmas 11 and 12 apply when understood about the behavior of the algorithm. This implies the following: the names are correct and execution terminates in $\mathcal{O}(n)$ time while $\mathcal{O}(n \log n)$ bits are used, all this with probability that is at least $1 - n^{-a}$. \square

Algorithm ARBITRARY-BOUNDED-MC is optimal with respect to the following performance measures: the expected time $\mathcal{O}(n)$, by Theorem 2, the expected number of random bits $\mathcal{O}(n \log n)$, by Proposition 1, and the probability of error $n^{-\mathcal{O}(1)}$, by Proposition 3.

9 Monte Carlo for Arbitrary with Unbounded Memory

We develop a Monte Carlo naming algorithm for Arbitrary PRAM with an unbounded amount of shared registers, when the number of processors n is unknown. The algorithm is called ARBITRARY-

Algorithm ARBITRARY-UNBOUNDED-MC

```
initialize  $k \leftarrow 1$                                 /* initial approximation of  $\lg n$  */
repeat
  initialize All-Named  $\leftarrow$  true
  initialize position $v$   $\leftarrow$  (0,0)
   $k \leftarrow r(k)$ 
  bin $v$   $\leftarrow$  random integer in  $[1, 2^k / (\beta k)]$     /* choose a bin for the ball */
  label $v$   $\leftarrow$  random integer in  $[1, 2^{\beta k}]$         /* choose a label for the ball */
  for  $i \leftarrow 1$  to  $\beta k$  do
    if position $v$  = (0,0) then
      Pad [bin $v$ ]  $\leftarrow$  label $v$ 
      if Pad [bin $v$ ] = label $v$  then
        Last-Name [bin $v$ ]  $\leftarrow$  Last-Name [bin $v$ ] + 1
        position $v$   $\leftarrow$  (bin $v$ , Last-Name [bin $v$ ])
  if position $v$  = (0,0) then
    All-Named  $\leftarrow$  false
until All-Named
name $v$   $\leftarrow$  the rank of position $v$ 
```

Figure 7: A pseudocode for a processor v of an Arbitrary PRAM, when the number of shared memory cells is unbounded. The variables **Pad** and **Last-Name** are arrays of shared memory cells, the variable **All-Named** is shared as well. The private variable **name** stores the acquired name. The constant $\beta > 0$ and an increasing function $r(k)$ are parameters.

UNBOUNDED-MC and its pseudocode is given in Figure 7.

The underlying idea is to parallelize the process of selection of names applied in Section 8 in algorithm ARBITRARY-BOUNDED-MC so that multiple processes could acquire information in the same round that later would allow them to obtain names. As algorithm ARBITRARY-BOUNDED-MC used shared registers **Pad** and **Last-Name**, the new algorithm uses arrays of shared registers playing similar roles. The values read-off from **Last-Name** cannot be used directly as names, because multiple processors can read the same values, so we need to distinguish between these values to assign names. To this end, we assign ranks to processors based on their lexicographic ordering by pairs of numbers determined by **Pad** and **Last-Name**.

The pseudocode in Figure 7 is structured as a repeat-loop. In the first iteration, the parameter k equals 1, and in subsequent ones is determined by iterations of the increasing integer-valued function $r(k)$, which is a parameter. We consider two instantiations of the algorithm, determined by $r(k) = k + 1$ and by $r(k) = 2k$. In one iteration of the main repeat-loop, a processor uses two

variables $\text{bin} \in [1, 2^k/(\beta k)]$ and $\text{label} \in [1, 2^{\beta k}]$, which are selected independently and uniformly at random from the respective ranges.

We interpret bin as a bin's number and label as a label for a ball. Processors write their values label into the respective bin by instruction $\text{Pad}[\text{bin}] \leftarrow \text{label}$ and verify what value got written. After a successful write, a processor increments $\text{Last-Name}[\text{bin}]$ and assigns the pair $(\text{bin}, \text{Last-Name}[\text{bin}])$ as its *position*. This is repeated βk times by way of iterating the inner for-loop. This loop has a specific upper bound βk on the number of iterations because we want to ascertain that there are at most βk balls in each bin. The main repeat-loop terminates when all values attempted to be written actually get written. Then processors assign themselves names according to the ranks of their positions. The array Last-Name is assumed to be initialized to 0's, and in each iteration of the repeat-loop we use a fresh region of shared memory to allocate this array.

Balls into bins. We consider a related process of placing labeled balls into bins, which is referred to as β -process. Such a process proceeds through stages and is parametrized by a function $r(k)$. In the first stage, we have $k = 1$, and given some value of k in a stage, the next stage has this parameter equal to $r(k)$. In a stage with a given k , we place n balls into $2^k/(\beta k)$ bins, with labels from $[1, 2^{\beta k}]$. The selections of bins and labels are performed independently and uniformly at random. A stage terminates the β -process when there are at most βk labels of balls in each bin.

Lemma 13 *The β -process always terminates.*

Proof: The β -process terminates by a stage in which the inequality $n \leq \beta k$ holds, because n is an upper bound on the number of balls in a bin. This always occurs when function $r(k)$ is increasing. \square

We expect the β -process to terminate earlier, as Lemma 14 states.

Lemma 14 *For each $a > 0$, if $k \leq \lg n - 2$ and $\beta \geq 1 + a$ then the probability of halting in the stage is smaller than n^{-a} , for sufficiently large n .*

Proof: We show that when k is suitably small then the probability of at most βk different labels in each bin is small. There are n balls placed into $2^k/(\beta k)$ bins, so there are at least $\frac{\beta k n}{2^k}$ balls in some bin, by the pigeonhole principle. We consider these balls and their labels.

The probability that all these balls have at most βk labels is at most

$$\begin{aligned} \binom{2^{\beta k}}{\beta k} \left(\frac{\beta k}{2^{\beta k}} \right)^{\frac{\beta k n}{2^k}} &\leq \left(\frac{e 2^{\beta k}}{\beta k} \right)^{\beta k} \cdot \frac{(\beta k)^{\frac{\beta k n}{2^k}}}{(2^{\beta k})^{\frac{\beta k n}{2^k}}} \\ &= e^{\beta k} 2^{\beta k(\beta k - \frac{\beta k n}{2^k})} (\beta k)^{\frac{\beta k n}{2^k} - \beta k} \\ &= e^{\beta k} \left(\frac{\beta k}{2^{\beta k}} \right)^{\frac{\beta k n}{2^k} - \beta k}. \end{aligned} \tag{10}$$

We want to show that this is at most n^{-a} . We compare the binary logarithms of n^{-a} and the right-hand side of (10), and want the following inequality to hold:

$$\lg e \cdot \beta k + \left(\frac{\beta k n}{2^k} - \beta k \right) (\lg(\beta k) - \beta k) \leq -a \lg n ,$$

which is equivalent to the following inequality, by algebra:

$$\frac{n}{2^k} \geq \frac{\lg e}{\beta k - \lg(\beta k)} + 1 + \frac{a \lg n}{\beta k(\beta k - \lg(\beta k))} . \quad (11)$$

Observe now that, assuming $\beta \geq a + 1$, if $k < \sqrt{\lg n}$ then the right-hand side of (11) is at most $\mathcal{O}(1) + \lg n$ while the left-hand side is at least \sqrt{n} , and when $\sqrt{\lg n} \leq k \leq \lg n - 2$ then the right-hand side of (11) is at most 3 while the left-hand side is at least 4, for sufficiently large n . \square

We say that a *label collision* occurs, in a configuration produced by the process, if some bin contains two balls with the same label.

Lemma 15 *For any $a > 0$, if $k > \frac{1}{2} \lg n$ and $\beta > 4a + 7$ then the probability of a label collision is smaller than n^{-a} .*

Proof: The number of pairs of a bin number and a label is $2^k \cdot 2^{\beta k} / (\beta k)$. It follows that the probability of some two balls in the same bin obtaining different labels is at least

$$\left(1 - \frac{n}{2^{k+\beta k} / (\beta k)}\right)^n \geq 1 - \frac{n^2}{2^{k+\beta k} / (\beta k)} ,$$

by the Bernoulli's inequality. So the probability that two different balls obtain the same label is at most $\frac{n^2}{2^{k+\beta k} / (\beta k)}$. We want the following inequality to hold

$$\frac{n^2}{2^{k+\beta k} / (\beta k)} < n^{-a} .$$

This is equivalent to the inequality obtained by taking logarithms

$$(2 + a) \lg n < (1 + \beta)k - \lg(\beta k) ,$$

which holds when $(2 + a) \lg n < \frac{1+\beta}{2}k$. It follows that it is sufficient for k to satisfy

$$k > \frac{2(2 + a)}{1 + \beta} \lg n .$$

This inequality holds for $k > \frac{1}{2} \lg n$ when $\beta > 4a + 7$. \square

We say that such a β -process is *correct* when upon termination no label collision occurs, otherwise the process is *incorrect*.

Lemma 16 *For any $a > 0$, there exists $\beta > 0$ such that the β -process is incorrect with probability that is at most n^{-a} , for sufficiently large n .*

Proof: The β -process is incorrect when there is a label collision after the last stage. The probability of the intersection of the events “ β -process terminates” and “there are label collisions” is bounded from above by the probability of any one of these events. Next we show that, for each pair of k and n , some of these two events occurs with probability that is at most n^{-a} , for a suitable β .

To this end we use Lemmas 14 and 15 in which we substitute $2a$ for a . We obtain that, on the one hand, if $k \leq \lg n - 2$ and $\beta \geq 1 + 2a$ then the probability of halting is smaller than n^{-2a} ,

and, on the other hand, that if $k > \frac{1}{2} \lg n$ and $\beta > 8a + 7$ then the probability of a label collision is smaller than n^{-2a} . It follows that some of the two considered events occurs with probability at most $2n^{-2a}$ for sufficiently large β and any sufficiently large n . This probability is at most n^{-a} , for sufficiently large n . \square

Lemma 17 *For any $a > 0$, there exists $\beta > 0$ and $c > 0$ such that the following two facts about the β -process hold. If $r(k) = k + 1$ then at most $cn/\ln n$ bins are ever needed and $cn \ln^2 n$ random bits are ever generated, each among these properties occurring with probability that is at least $1 - n^{-a}$. If $r(k) = 2k$ then at most $cn^2/\ln n$ bins are ever needed and $cn \ln n$ random bits are ever generated, each among these properties occurring with probability that is at least $1 - n^{-a}$.*

Proof: We throw n balls into $2^k/(\beta k)$ bins. As k keeps increasing, the probability of termination increases as well, because both $2^k/(\beta k)$ and βk increase as functions of k . Let us take $k = 1 + \lg n$ so that the number of bins is $\frac{2n}{\beta k}$. We want to show that no bin contains more than βk balls with a suitably small probability.

Let us consider a specific bin and let X be the number of balls in this bin. The expected number of balls in the bin is $\mu = \frac{\beta k}{2}$. We use the Chernoff bound for a sequence of Bernoulli trials in the form of

$$\Pr(X > (1 + \varepsilon)\mu) < \exp(-\varepsilon^2\mu/3) ,$$

which holds for $0 < \varepsilon < 1$, see [52]. Let us choose $\varepsilon = \frac{1}{2}$, so that $1 + \varepsilon = \frac{3}{2}$ and $\frac{3}{2}\mu = \frac{3}{4}\beta k$. We obtain the following bound

$$\Pr(X > \beta k) < \Pr(X > \frac{3}{4} \cdot \beta k) < \exp(-\frac{1}{4} \cdot \frac{\beta k}{6}) = \exp(-\frac{\beta}{24} \cdot (1 + \lg n)) ,$$

which can be made smaller than n^{-1-a} for a β sufficiently large with respect to a , and sufficiently large n . Using the union bound, each of the n bins contains at most βk balls with probability at most n^{-a} . This implies that termination occurs as soon as k reaches or surpasses $k = 1 + \lg n$, with the corresponding large probability $1 - n^{-a}$.

In the case of $r(k) = k + 1$, the consecutive integer values of k are tried, so the β -process terminates by the time $k = 1 + \lg n$, and for this k the number of bins needed is $\Theta(n/\log n)$. To choose a bin for any value of k requires at most k random bits, so implementing such choices for $k = 1, 2, \dots, 1 + \lg n$ requires $\mathcal{O}(\log^2 n)$ random bits per processor.

In the case of $r(k) = 2k$, the β -process terminates by the time the magnitude of k reaches $2(1 + \lg n)$, and for this value of k the number of bins needed is $\Theta(n^2/\log n)$. As k progresses through consecutive powers of 2, the sum of these numbers is a sum of a geometric progression, and so is of the order of the maximum term, that is $\Theta(\log n)$, which is the number of random bits per processor. \square

There is a direct correspondence between iterations of the outer repeat-loop of algorithm ARBITRARY-UNBOUNDED-MC and stages of the β -process. We map an execution of the algorithm into a corresponding execution of a β -process in order to apply Lemmas 16 and 17 in the proof of the following Theorem, which summarizes the performance of algorithm ARBITRARY-UNBOUNDED-MC and justifies that it is Monte Carlo.

Theorem 9 *Algorithm ARBITRARY-UNBOUNDED-MC always terminates, for any $\beta > 0$. For each $a > 0$, there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns unique names and has the following additional properties with probability $1 - n^{-a}$. If $r(k) = k + 1$ then at most $cn/\ln n$ memory cells are ever needed, $cn \ln^2 n$ random bits are ever generated, and the algorithm terminates in time $\mathcal{O}(\log^2 n)$. If $r(k) = 2k$ then at most $cn^2/\ln n$ memory cells are ever needed, $cn \ln n$ random bits are ever generated, and the algorithm terminates in time $\mathcal{O}(\log n)$.*

Proof: The algorithm always terminates by Lemma 13. By Lemma 16, the algorithm assigns correct names with probability that is at least $1 - n^{-a}$. The remaining properties follow from Lemma 17, because the number of bins is proportional to the number of memory cells and the number of random bits per processor is proportional to time. \square

The instantiations of algorithm ARBITRARY-UNBOUNDED-MC are close to optimality with respect to some of the performance metrics we consider, depending on whether $r(k) = k + 1$ or $r(k) = 2k$. If $r(k) = k + 1$ then the algorithm's use of shared memory would be optimal if its time were $\mathcal{O}(\log n)$, by Theorem 2, but as it is, the algorithm misses space optimality by at most a logarithmic factor, since the algorithm's running time is $\mathcal{O}(\log^2 n)$. Similarly, if $r(k) = k + 1$ then the number of random bits ever generated $\mathcal{O}(n \log^2 n)$ misses optimality by at most a logarithmic factor, by Proposition 1. On the other hand, if $r(k) = 2k$ then the expected time $\mathcal{O}(\log n)$ is optimal, by Theorem 3, the expected number of random bits $\mathcal{O}(n \log n)$ is optimal, by Proposition 1, and the probability of error $n^{-\mathcal{O}(1)}$ is optimal, by Proposition 3, but the amount of used shared memory misses optimality by at most a polynomial factor, by Theorem 2.

10 Monte Carlo for Common with Bounded Memory

The Monte Carlo algorithm COMMON-BOUNDED-MC, which we present in this section, solves the naming problem for a Common PRAM with a constant number of shared read-write registers, when the number of processors n is unknown. The algorithm has its pseudocode in Figure 10. To make the exposition of this algorithm more modular, we use two procedures ESTIMATE-SIZE and EXTEND-NAMES. The pseudocodes of these procedures are given in Figures 8 and 9, respectively. The private variables in the pseudocode in Figure 10 have the following meaning: **size** is an approximation of the number of processors n , and **number-of-bins** determines the size of the range of bins we throw conceptual balls into.

The main task of procedure ESTIMATE-SIZE is to produce an estimate of the number n of processors. Procedure EXTEND-NAMES is iterated multiple times, each iteration is intended to assign names to a group of processors. This is accomplished by the processors selecting integer values at random, interpreted as throwing balls into bins, and verifying for collisions. Each selection of a bin is followed by a collision detection. A ball placement without a detected collision results in a name assigned, otherwise the involved processors try again to throw balls into a range of bins. The effectiveness of the resulting algorithm hinges of calibrating the number of bins to the expected number of balls to be thrown.

Balls into bins for the first time. The role of procedure ESTIMATE-SIZE, when called by algorithm COMMON-BOUNDED-MC, is to estimate the unknown number of processors n , which is returned as **size**, to assign a value to variable **number-of-bins**, and assign values to each private variable **bin**, which indicates the number of a selected bin in the range $[1, \text{number-of-bins}]$. The

Procedure ESTIMATE-SIZE

```

initialize  $k \leftarrow 2$                                 /* initial approximation of  $\lg n$  */

repeat
   $k \leftarrow k + 1$ 
   $\text{bin}_v \leftarrow$  random integer in  $[1, k 2^k]$ 
  initialize Nonempty-Bins  $\leftarrow 0$ 
  for  $i \leftarrow 1$  to  $k 2^k$  do
    if  $\text{bin}_v = i$  then
      Nonempty-Bins  $\leftarrow$  Nonempty-Bins + 1

until Nonempty-Bins  $\leq 2^k$ 

return  $(3 \cdot 2^k, k 2^k)$                                /*  $3 \cdot 2^k$  is size,  $k 2^k$  is number-of-bins */

```

Figure 8: A pseudocode for a processor v of a Common PRAM. This procedure is invoked by algorithm COMMON-BOUNDED-MC in Figure 10. The variable Nonempty-Bins is shared.

procedure tries consecutive values of k as approximations of $\lg n$. For a given k , an experiment is carried out to throw n balls into $k 2^k$ bins. The execution stops when the number of occupied bins is at most 2^k , and then $3 \cdot 2^k$ is treated as an approximation of n and $k 2^k$ is the returned number of bins.

Lemma 18 *For $n \geq 20$ processors, procedure ESTIMATE-SIZE returns an estimate **size** of n such that the inequality $\text{size} < 6n$ holds with certainty and the inequality $n < \text{size}$ holds with probability $1 - 2^{-\Omega(n)}$.*

Proof: The procedure returns $3 \cdot 2^k$, for some integer $k > 0$. We interpret selecting of values for variable **bin** in an iteration of the main repeat-loop as throwing n balls into $k 2^k$ bins; here $k = j + 2$ in the j th iteration of this loop, because the smallest value of k is 3. Clearly, n is an upper bound on the number of occupied bins.

If n is a power of 2, say $n = 2^i$, then the procedure terminates by the time $i = k$, so that $2^k < 2^{i+1} = 2n$. Otherwise, the maximum possible k equals $\lceil \lg n \rceil$, because $2^{\lceil \lg n \rceil} < n < 2^{\lceil \lg n \rceil + 1}$. This gives $2^{\lceil \lg n \rceil} = 2^{\lceil \lg n \rceil + 1} < 2n$. We obtain that the inequality $2^k < 2n$ occurs with certainty, and so does $3 \cdot 2^k < 6n$ as well.

Now we estimate the lower bound on 2^k . Consider k such that $2^k \leq \frac{n}{3}$. Then n balls fall into at most 2^k bins with probability that is at most

$$\binom{k 2^k}{2^k} \left(\frac{2^k}{k 2^k} \right)^n \leq \left(\frac{e k 2^k}{2^k} \right)^{2^k} \cdot \frac{1}{k^n} = (e k)^{2^k} k^{-n} = e^{2^k} k^{2^k - n} \leq e^{n/3} k^{-2n/3}. \quad (12)$$

The right-hand side of (12) is at most $e^{-n/3}$ when the inequality $k > e$ holds. The smallest k considered in the pseudocode in Figure 8 is $k = 3 > e$. The inequality $k > e$ is consistent with

Procedure EXTEND-NAMES

```

initialize Collision-Detected  $\leftarrow$  collisionv  $\leftarrow$  false

for  $i \leftarrow 1$  to number-of-bins do

  if binx =  $i$  for some processor  $x$  then
    if binv =  $i$  then
      for  $j \leftarrow 1$  to  $\beta \lg \text{size}$  do
        if VERIFY-COLLISION then
          Collision-Detected  $\leftarrow$  collisionv  $\leftarrow$  true
      if not collisionv then
        Last-Name  $\leftarrow$  Last-Name + 1
        namev  $\leftarrow$  Last-Name
        binv  $\leftarrow$  0

  if (number-of-bins > size) then
    number-of-bins  $\leftarrow$  size

  if collisionv then
    binv  $\leftarrow$  random integer in  $[1, \text{number-of-bins}]$ 

```

Figure 9: A pseudocode for a processor v of a Common PRAM. This procedure invokes procedure VERIFY-COLLISION, whose pseudocode is in Figure 1, and is itself invoked by algorithm COMMON-BOUNDED-MC in Figure 10. The variables **Last-Name** and **Collision-Detected** are shared. The private variable **name** stores the acquired name. The constant $\beta > 0$ is to be determined in analysis.

$2^k \leq \frac{n}{3}$ when $n \geq 20$. The number of possible values for k is $\mathcal{O}(\log n)$, so the probability of the procedure returning for $2^k \leq \frac{n}{3}$ is $e^{-n/3} \cdot \mathcal{O}(\log n) = 2^{-\Omega(n)}$. \square

Procedure EXTEND-NAMES's behavior can also be interpreted as throwing balls into bins, where a processor v 's ball is in a bin x when **bin**_v = x . The procedure first verifies the suitable range of bins $[1, \text{number-of-bins}]$ for collisions. A verification for collisions takes either just a constant time or $\Theta(\log n)$ time.

A constant verification occurs when there is no ball in the considered bin i , which is verified when the line “if **bin**_x = i for some processor x ” in the pseudocode in Figure 9 is to be executed. Such a verification is performed by using a shared register initialized to 0, into which all processors v with **bin**_v = i write 1, then all the processors read this register, and if the outcome of reading is 1 then all write 0 again, which indicates that there is at least one ball in the bin, otherwise there is no ball.

A logarithmic-time verification of collision occurs when there is some ball in the corresponding bin. This triggers calling procedure VERIFY-COLLISION precisely $\beta \lg n$ times; notice that this

procedure has the default parameter 1, as only one bin is verified at a time. Ultimately, when a collision is not detected for some processor v whose ball is the bin, then this processor increments **Last-Name** and assigns its new value as a tentative name. Otherwise, when a collision is detected, processor v places its ball in a new bin when the last line in Figure 9 is executed.

To prepare for the next round of throwing balls, the variable **number-of-bins** may be reset. During one iteration of the main repeat-loop of the pseudocode of algorithm COMMON-BOUNDED-MC in Figure 10, the number of bins is first set to a value that is $\Theta(n \log n)$ by procedure ESTIMATE-SIZE. Immediately after that, it is reset to $\Theta(n)$ by the first call of procedure EXTEND-NAMES, in which the instruction **number-of-bins** \leftarrow **size** is performed. Here, we need to notice that **number-of-bins** $= \Theta(n \log n)$ and **size** $= \Theta(n)$, by the pseudocodes in Figures 8 and 10 and Lemma 18.

Balls into bins for the second time. In the course of analysis of performance of procedure EXTEND-NAMES, we consider a balls-into-bins process; we call it simply the *ball process*. It proceeds through stages so that in a stage we have a number of balls which we throw into a number of bins. The sets of bins used in different stages are disjoint. The number of balls and bins used in a stage are as determined in the pseudocode in Figure 9, which means that there are n balls and the numbers of bins are as determined by an execution of procedure ESTIMATE-SIZE, that is, the first stage uses **number-of-bins** bins and subsequent stages use **size** bins, as returned by ESTIMATE-SIZE.

The only difference between the ball process and the actions of procedure EXTEND-NAMES is that collisions are detected with certainty in the ball process rather than being tested for. In particular, the parameter β is not involved in the ball process (nor in its name). The ball process terminates in the first stage in which no multiple bins are produced, so that there are no collisions among the balls.

Lemma 19 *The ball process results in all balls ending single in their bins and the number of times a ball is thrown, summed over all the stages, being $\mathcal{O}(n)$, both events occurring with probability $1 - n^{-\Omega(\log n)}$.*

Proof: The argument leverages the property that, in each stage, the number of bins exceeds the number of balls by at least a logarithmic factor. We will denote the number of bins in a stage by m . This number will take on two values, first $m = k2^k$ returned as **number-of-bins** by procedure ESTIMATE-SIZE and then $m = 3 \cdot 2^k$ returned as **size** by the same procedure ESTIMATE-SIZE, for $k > 3$. Because $m = k2^k$ in the first stage, and also **size** $= 3 \cdot 2^k > n$, by Lemma 18, we obtain that $m > \frac{n}{3} \lg \frac{n}{3}$ in the first stage, and that m is at least n in the following stages, with probability exponentially close to 1.

In the first stage, we throw $\ell_1 = n$ balls into at least $m = \frac{n}{3} \lg \frac{n}{3}$ bins, with large probability. Conditional on the event that there are at least these many bins, the probability that a given ball ends the stage single in a bin is

$$m \cdot \frac{1}{m} \left(1 - \frac{1}{m}\right)^{\ell_1 - 1} \geq 1 - \frac{\ell_1 - 1}{m} \geq 1 - \frac{n - 1}{\frac{n}{3} \lg \frac{n}{3}} \geq 1 - \frac{4}{\lg n},$$

for sufficiently large n , where we used the Bernoulli's inequality. Let Y_1 be the number of singleton bins in the first stage. The expectancy of Y_1 satisfies

$$\mathbb{E}[Y_1] \geq \ell_1 \left(1 - \frac{4}{\lg n}\right).$$

To estimate the deviation of Y_1 from its expected value $\mathbb{E}[Y_1]$ we use the bounded differences inequality [49, 52]. Let B_j be the bin of ball b_j , for $1 \leq j \leq \ell_1$. Then Y_1 is of the form $Y_1 = h(B_1, \dots, B_{\ell_1})$, where h satisfies the Lipschitz condition with constant 2, because moving one ball to a different bin results in changing the value of h by at most 2 with respect to the original value. The bounded-differences inequality specialized to this instance is as follows, for any $d > 0$:

$$\Pr(Y_1 \leq \mathbb{E}[Y_1] - d\sqrt{\ell_1}) \leq \exp(-d^2/8) . \quad (13)$$

We employ $d = \lg n$, which makes the right-hand side of (13) asymptotically equal to $n^{-\Omega(\log n)}$. The number of balls ℓ_2 eligible for the second stage can be estimated as follows, this bound holding with probability $1 - n^{-\Omega(\log n)}$:

$$\ell_2 \leq \frac{4\ell_1}{\lg n} + \lg n \sqrt{\ell_1} = \frac{4\ell_1}{\lg n} \left(1 + \frac{\lg^2 n}{4\sqrt{\ell_1}}\right) \leq \frac{5n}{\lg n} , \quad (14)$$

for sufficiently large n .

In the second stage, we throw ℓ_2 balls into $m \geq n$ bins, with large probability. Conditional on the bound (14) holding, the probability that a given ball ends up single in a bin is

$$m \cdot \frac{1}{m} \left(1 - \frac{1}{m}\right)^{\ell_2 - 1} \geq 1 - \frac{\ell_2 - 1}{m} \geq 1 - \frac{5}{\lg n} ,$$

where we used the Bernoulli's inequality. Let Y_2 be the number of singleton bins in the second stage. The expectancy of Y_2 satisfies

$$\mathbb{E}[Y_2] \geq \ell_2 \left(1 - \frac{5}{\lg n}\right) .$$

To estimate the deviation of Y_2 from its expected value $\mathbb{E}[Y_2]$, we again use the bounded differences inequality, which specialized to this instance is as follows, for any $d > 0$:

$$\Pr(Y_2 \leq \mathbb{E}[Y_2] - d\sqrt{\ell_2}) \leq \exp(-d^2/8) . \quad (15)$$

We again employ $d = \lg n$, which makes the right-hand side of (15) asymptotically equal to $n^{-\Omega(\log n)}$. The number of balls ℓ_3 eligible for the third stage can be bounded from above as follows, which holds with probability $1 - n^{-\Omega(\log n)}$,

$$\ell_3 \leq \frac{5\ell_2}{\lg n} + \lg n \sqrt{\ell_2} = \frac{5\ell_2}{\lg n} \left(1 + \frac{\lg^2 n}{5\sqrt{\ell_2}}\right) \leq \frac{6n}{\lg^2 n} , \quad (16)$$

for sufficiently large n .

Next, we generalize these estimates. In stages i , for $i \geq 2$, among the first $\mathcal{O}(\log n)$ ones, we throw balls into $m \geq n$ bins with large probability. Let ℓ_i be the number of balls eligible for such a stage i . We show by induction that ℓ_i , for $i \geq 3$, can be estimated as follows:

$$\ell_i \leq \frac{6n}{\lg^2 n} \cdot 4^{3-i} \quad (17)$$

with probability $1 - n^{-\Omega(\log n)}$. The estimate (16) provides the base of induction for $i = 3$. In the inductive step, we assume (17), and consider what happens during stage $i > 3$ in order to estimate the number of balls eligible for the next stage $i + 1$.

In stage i , we throw ℓ_i balls into $m \geq n$ bins, with large probability. Conditional on the bound (17), the probability that a given ball ends up single in a bin is

$$m \cdot \frac{1}{m} \left(1 - \frac{1}{m}\right)^{\ell_i-1} \geq 1 - \frac{\ell_i - 1}{m} \geq 1 - \frac{6 \cdot 4^{3-i}}{\lg^2 n},$$

by the inductive assumption, where we also used the Bernoulli's inequality. If Y_i is the number of singleton bins in stage i , then its expectation $\mathbb{E}[Y_i]$ satisfies

$$\mathbb{E}[Y_i] \geq \ell_i \left(1 - \frac{6 \cdot 4^{3-i}}{\lg^2 n}\right). \quad (18)$$

To estimate the deviation of Y_i from its expected value $\mathbb{E}[Y_i]$, we again use the bounded differences inequality, which specialized to this instance is as follows, for any $d > 0$:

$$\Pr(Y_i \leq \mathbb{E}[Y_i] - d\sqrt{\ell_i}) \leq \exp(-d^2/8). \quad (19)$$

We employ $d = \lg n$, which makes the right-hand side of (19) asymptotically equal to $n^{-\Omega(\lg n)}$. The number of balls ℓ_{i+1} eligible for the next stage $i+1$ can be estimated from above in the following way, the estimate holding with probability $1 - n^{-\Omega(\lg n)}$:

$$\begin{aligned} \ell_{i+1} &\leq \frac{6 \cdot 4^{3-i} \cdot \ell_i}{\lg^2 n} + \lg n \sqrt{\ell_i} \\ &= \frac{6 \cdot 4^{3-i} \cdot \ell_i}{\lg^2 n} \left(1 + \frac{1}{6} 4^{i-3} \lg^3 n \cdot \ell_i^{-1/2}\right) \\ &\leq \frac{6 \cdot 4^{3-i}}{\lg^2 n} \cdot \frac{6n}{\lg^2 n} \cdot 4^{3-i} \cdot \left(1 + \frac{4^{(i-3)/2} \lg^4 n}{6\sqrt{6n}}\right) \\ &\leq \frac{6n}{\lg^2 n} \cdot 4^{3-i} \cdot \left(\frac{6 \cdot 4^{3-i}}{\lg^2 n} + \frac{4^{(3-i)/2} \lg^2 n}{\sqrt{6n}}\right) \\ &\leq \frac{6n}{\lg^2 n} \cdot 4^{3-i} \cdot \left(\frac{6}{\lg^2 n} + \frac{\lg^2 n}{\sqrt{6n}}\right) \\ &\leq \frac{6n}{\lg^2 n} \cdot 4^{3-i-1}, \end{aligned}$$

for sufficiently large n that does not depend on i . For the event $Y_i \leq \mathbb{E}[Y_i] - d\sqrt{\ell_i}$ in the estimate (19) to be meaningful, it is sufficient if the following estimate holds:

$$\lg n \cdot \sqrt{\ell_i} = o(\mathbb{E}[Y_i]).$$

This is the case as long as $\ell_i > \lg^3 n$, because $\mathbb{E}[Y_i] = \ell_i(1 + o(1))$ by (18).

To summarize at this point, as long as ℓ_i is sufficiently large, that is, $\ell_i > \lg^3 n$, the number of eligible balls decreases by at least a factor of 4 with probability that is at least $1 - n^{-\Omega(\lg n)}$. It follows that the total number of eligible balls, summed over these stages, is $\mathcal{O}(n)$ with this probability.

After at most $\lg_4 n = \frac{1}{2} \lg n$ such stages, the number of balls becomes at most $\lg^3 n$ with probability $1 - n^{-\Omega(\lg n)}$. This number of stages is half of the number of times the for-loop is iterated in the pseudocode in Figure 10.

Algorithm COMMON-BOUNDED-MC

```
repeat
  initialize Last-Name  $\leftarrow 0$ 
  (size, number-of-bins)  $\leftarrow$  ESTIMATE-SIZE
  for  $\ell \leftarrow 1$  to  $\lg \text{size}$  do
    EXTEND-NAMES
  if not Collision-Detected then return
```

Figure 10: A pseudocode for a processor v of a Common PRAM, where there is a constant number of shared memory cells. Procedures ESTIMATE-SIZE and EXTEND-NAMES have their pseudocodes in Figures 8 and 9, respectively. The variables Last-Name and Collision-Detected are shared.

It remains to consider the stages when $\ell_i \leq \lg^3 n$, so that we throw at most $\lg^3 n$ balls into at least n bins. They all end up in singleton bins with a probability that is at least

$$\left(\frac{n - \lg^3 n}{n}\right)^{\lg^3 n} \geq \left(1 - \frac{\lg^3 n}{n}\right)^{\lg^3 n} \geq 1 - \frac{\lg^6 n}{n},$$

by the Bernoulli's inequality. So the probability of a collision is at most $\frac{\lg^6 n}{n}$. One stage without any collision terminates the process. If we repeat such stages $\frac{1}{2} \lg n$ times, without even removing single balls, then the probability of collisions occurring in all these stages is at most

$$\left(\frac{\lg^6 n}{n}\right)^{\frac{1}{2} \lg n} = n^{-\Omega(\log n)}.$$

This number of stages is half of the number of times the for-loop is iterated in the pseudocode in Figure 10. The number of eligible balls summed over these final stages is at most $\lg^7 n = o(n)$. \square

The following Theorem 10 summarizes the performance of algorithm COMMON-BOUNDED-MC (see the pseudocode in Figure 10) as a Monte Carlo one.

Theorem 10 *Algorithm COMMON-BOUNDED-MC terminates almost surely. For each $a > 0$, there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns unique names, works in time at most $cn \ln n$, and uses at most $cn \ln n$ random bits, each among these properties holding with probability at least $1 - n^{-a}$.*

Proof: One iteration of the main repeat-loop suffices to assign names with probability $1 - n^{-\Omega(\log n)}$, by Lemma 19. This means that the probability of not terminating by the i th iteration is at most $(n^{-\Omega(\log n)})^i$, which converges to 0 with i growing to infinity.

The algorithm returns duplicate names only when a collision occurs that is not detected by procedure VERIFY-COLLISION. For a given multiple bin, one iteration of this procedure does not detect collision with probability at most $1/2$, by Lemma 1. Therefore $\beta \lg \text{size}$ iterations do not detect collision with probability $\mathcal{O}(n^{-\beta/2})$, by Lemma 18. The number of nonempty bins ever tested

is at most dn , for some constant $d > 0$, by Lemma 19, with the suitably large probability. Applying the union bound results in the estimate n^{-a} on the probability of error for sufficiently large β .

The duration of an iteration of the inner for-loop is either constant, then we call it *short*, or it takes time $\mathcal{O}(\log \text{size})$, then we call it *long*. First, we estimate the total time spent on short iterations. This time in the first iteration of the inner for-loop is proportional to **number-of-bins** returned by procedure ESTIMATE-SIZE, which is at most $6n \cdot \lg(6n)$, by Lemma 18. Each of the subsequent iterations takes time proportional to **size**, which is at most $6n$, again by Lemma 18. We obtain that the total number of short iterations is $\mathcal{O}(n \log n)$ in the worst case. Next, we estimate the total time spent on long iterations. One such an iteration has time proportional to $\lg \text{size}$, which is at most $\lg 6n$ with certainty. The number of such iterations is at most dn with probability $1 - n^{-\Omega(\log n)}$, for some constant $d > 0$, by Lemma 19. We obtain that the total number of long iterations is $\mathcal{O}(n \log n)$, with the correspondingly large probability. Combining the estimates for short and long iterations, we obtain $\mathcal{O}(n \log n)$ as a bound on time of one iteration of the main repeat-loop. One such an iteration suffices with probability $1 - n^{-\Omega(\log n)}$, by Lemma 19.

Throwing one ball uses $\mathcal{O}(\log n)$ random bits, by Lemma 18. The number of throws is $\mathcal{O}(n)$ with the suitably large probability, by Lemma 19. \square

Algorithm COMMON-BOUNDED-MC is optimal with respect to the following performance metrics: the expected time $\mathcal{O}(n \log n)$, by Theorem 1, the number of random bits $\mathcal{O}(n \log n)$, by Proposition 1, and the probability of error $n^{-\mathcal{O}(1)}$, by Proposition 3.

11 Monte Carlo for Common with Unbounded Memory

We consider naming on a Common PRAM in the case when the amount of shared memory is unbounded and the number of processors n is unknown. The Monte Carlo algorithm we propose, called COMMON-UNBOUNDED-MC, is similar to algorithm COMMON-BOUNDED-MC in Section 10, in that it involves a randomized experiment to estimate the number of processors of the PRAM. Such an experiment is then followed by repeatedly throwing balls into bins, testing for collisions, and throwing again if a collision is detected, until eventually no collisions are detected.

Algorithm COMMON-UNBOUNDED-MC has its pseudocode given in Figure 12. The algorithm is structured as a repeat loop. An iteration starts by invoking procedure GAUGE-SIZE, whose pseudocode is in Figure 11. This procedure returns **size** as an estimate of the number of processors n . Next, a processor chooses randomly a bin in the range $[1, 3\text{size}]$. Then it keeps verifying for collisions $\beta \lg \text{size}$, in such a manner that when a collision is detected then a new bin is selected from the same range. After such $\beta \lg \text{size}$ verifications and possible new selections of bins, another $\beta \lg \text{size}$ verifications follow, but without changing the selected bins. When no collision is detected in the second segment of $\beta \lg \text{size}$ verifications, then this terminates the repeat-loop, which triggers assigning each station the rank of the selected bin, by a prefix-like computation. If a collision is detected in the second segment of $\beta \lg \text{size}$ verifications, then this starts another iteration of the main repeat-loop.

Procedure GAUGE-SIZE-MC returns an estimate of the number n of processors in the form 2^k , for some positive integer k . It operates by trying various values of k , and, for a considered k , by throwing n balls into 2^k bins and next counting how many bins contain balls. Such counting is performed by a prefix-like computation, whose pseudocode is omitted in Figure 11. The additional

Procedure GAUGE-SIZE-MC

```

 $k \leftarrow 1$ 

repeat
     $k \leftarrow r(k)$ 
     $\text{bin}_v \leftarrow$  random integer in  $[1, 2^k]$ 

until the number of selected values of variable  $\text{bin}$  is  $\leq 2^k/\beta$ 

return  $(\lceil 2^{k+1}/\beta \rceil)$ 

```

Figure 11: A pseudocode for a processor v of a Common PRAM, where the number of shared memory cells is unbounded. The constant $\beta > 0$ is the same parameter as in Figure 12, and an increasing function $r(k)$ is also a parameter.

parameter $\beta > 0$ is a number that affects the probability of underestimating n .

The way in which selections of numbers k is performed is controlled by function $r(k)$, which is a parameter. We will consider two instantiations of this function: one is function $r(k) = k + 1$ and the other is function $r(k) = 2k$.

Lemma 20 *If $r(k) = k + 1$ then the value of size as returned by GAUGE-SIZE-MC satisfies $\text{size} \leq 2n$ with certainty and the inequality $\text{size} \geq n$ holds with probability $1 - \beta^{-n/3}$.*

If $r(k) = 2k$ then the value of size as returned by GAUGE-SIZE-MC satisfies $\text{size} \leq 2\beta n^2$ with certainty and $\text{size} \geq \beta n^2/2$ with probability $1 - \beta^{-n/3}$.

Proof: We model procedure's execution by an experiment of throwing n balls into 2^k bins. If the parameter function $r(k)$ is $r(k) = k + 1$ then this results in trying all possible consecutive values of k , starting from $k = 2$, so that $k = i + 1$ in the i th iteration of the repeat-loop. If the parameter function $r(k)$ is $r(k) = 2k$ then k takes on only the powers of 2.

There are at most n bins occupied in any such an experiment. Therefore, the procedure returns by the time the inequality $2^k/\beta \geq n$ holds, where k determines the range of bins. It follows that if $r(k) = k + 1$ then the returned value $\lceil 2^{k+1}/\beta \rceil$ is at most $2n$. If $r(k) = 2k$ then the worst error in estimating occurs when $2^i/\beta = n - 1$ for some i that is a power of 2. Then the returned value is $2^{2i}/\beta = (\beta(n - 1))^2/\beta$, which is at most $2\beta n^2$, this occurring with probability $1 - \beta^{-n/3}$.

Given 2^k bins, we estimate the probability that the number of occupied bins is at most $2^k/\beta$. It is

$$\binom{2^k}{2^k/\beta} \left(\frac{2^k/\beta}{2^k} \right)^n \leq \left(\frac{2^k e}{2^k/\beta} \right)^{2^k/\beta} \cdot \frac{1}{\beta^n} = (e\beta)^{2^k/\beta} \cdot \beta^{-n}.$$

Next, we identify a range of values of k for which this probability is exponentially close to 0 with respect to n .

To this end, let $0 < \rho < 1$ and let us consider the inequality

$$(e\beta)^{2^k/\beta} \cdot \beta^{-n} < \rho^n. \tag{20}$$

It is equivalent to the following one

$$\frac{2^k}{\beta}(1 + \ln \beta) - n \ln \beta < n \ln \rho ,$$

by taking logarithms of both sides. This in turn is equivalent to

$$\frac{2^k}{\beta}(1 + \ln \beta) < n \left(\ln \beta - \ln \frac{1}{\rho} \right) . \quad (21)$$

Let us choose $\rho = \beta^{-1/2}$ in (21). Then (20) specialized to this particular ρ is equivalent to the following inequality

$$\frac{2^k}{\beta}(1 + \ln \beta) < n \frac{\ln \beta}{2} .$$

This in turn leads to the estimate

$$2^k < n \cdot \frac{\ln \beta}{2} \cdot \frac{\beta}{1 + \ln \beta} < \frac{\beta}{2} \cdot n ,$$

which means $2^{k+1}/\beta < n$. When k satisfies this inequality then the probability of returning is at most $\beta^{-n/2}$. There are $\mathcal{O}(\log n)$ such values of k considered by the procedure, so it returns for one of them with probability at most

$$\mathcal{O}(\log n) \cdot \beta^{-n/2} < \beta^{-n/3} ,$$

for sufficiently large n . Therefore, with probability at least $1 - \beta^{-n/3}$, the returned value $\lceil 2^{k+1}/\beta \rceil$ is at least as large as determined the first considered k that satisfies $2^{k+1}/\beta \geq n$.

If $r(k) = k + 1$ then all the possible exponents k are considered, so the returned value $\lceil 2^{k+1}/\beta \rceil$ is at least n with probability $1 - \beta^{-n/3}$. If $r(k) = 2k$ then the worst error of estimating n occurs when $2^{i+1}/\beta = n - 1$ for some i that is a power of 2. Then the returned value is

$$2^{2i+1}/\beta = 2 \cdot (\beta(n - 1)/2)^2/\beta ,$$

which is at least $\beta n^2/2$, this occurring with probability $1 - \beta^{-n/3}$. □

We discuss performance of algorithm COMMON-UNBOUNDED-MC (see the pseudocode in Figure 12) by referring to analysis of a related algorithm COMMON-UNBOUNDED-LV given in Section 7. We consider a β -process with verifications, which is defined as follows. The process proceeds through stages. The first stage starts with placing n balls into `3size` bins. For each of the subsequent stages, for all multiple bins and for each ball in such a bin, we perform a Bernoulli trial with the probability $\frac{1}{2}$ of success, which represents the outcome of procedure VERIFY-COLLISION. A success in a trial is referred to as a *positive verification* otherwise it is a *negative* one. If at least one positive verification occurs for a ball in a multiple bin then all the balls in this bin are relocated in this stage to bins selected uniformly at random and independently for each such a ball, otherwise the balls stay put in this bin until the next stage. The process terminates when all balls are single in their bins.

Lemma 21 *For any number $a > 0$, there exists $\beta > 0$ such that the β -process with verifications terminates within $\beta \lg n$ stages with all of them comprising the total of $\mathcal{O}(n)$ ball throws with probability at least $1 - n^{-a}$.*

Algorithm COMMON-UNBOUNDED-MC

```

repeat
    size ← GAUGE-SIZE
    binv ← random integer in [1, 3size]
    for i ← 1 to β lg size do
        if VERIFY-COLLISION(binv) then
            binv ← random number in [1, 3size]
    Collision-Detected ← false
    for i ← 1 to β lg size do
        if VERIFY-COLLISION(binv) then
            Collision-Detected ← true
until not Collision-Detected
namev ← the rank of binv among selected bins

```

Figure 12: A pseudocode for a processor v of a Common PRAM, where the number of shared memory cells is unbounded. The constant $\beta > 0$ is a parameter impacting the probability of error. The private variable **name** stores the acquired name.

Proof: We use the analysis of a ball process relevant to Common PRAM with unbounded memory given in Section 7. The constant 3 determining our β -process with verifications corresponds to $1 + \beta$ in Section 7. The corresponding β -process in verifications considered in Section 7 is defined by referring to known n . We use the approximation **size** instead, which is at least as large as n with probability $1 - \beta^{-n/3}$, by Lemma 20 just proved. By Lemma 10, our β -process with verifications does not terminate within $\beta \lg n$ stages when **size** $\geq n$ with probability at most n^{-2a} and the inequality **size** $\geq n$ does not hold with probability at most $\beta^{-n/3}$. Therefore the conclusion we want to prove does not hold with probability at most $n^{-2a} + \beta^{-n/3}$, which is at most n^{-2a} for sufficiently large n . \square

The following Theorem summarizes the performance of algorithm COMMON-UNBOUNDED-MC (see the pseudocode in Figure 12) as a Monte Carlo one. Its proof relies on mapping an execution of the β -process with verifications on executions of algorithm COMMON-UNBOUNDED-MC in a natural manner.

Theorem 11 *Algorithm COMMON-UNBOUNDED-MC terminates almost surely, for a sufficiently large β . For each $a > 0$, there exists $\beta > 0$ and $c > 0$ such that the algorithm assigns unique names and has the following additional properties with probability $1 - n^{-a}$. If $r(k) = k + 1$ then at most cn memory cells are ever needed, $cn \ln^2 n$ random bits are ever generated, and the algorithm terminates in time $\mathcal{O}(\log^2 n)$. If $r(k) = 2k$ then at most cn^2 memory cells are ever needed, $cn \ln n$ random bits are ever generated, and the algorithm terminates in time $\mathcal{O}(\log n)$.*

Proof: For a given $a > 0$, let us take β that exists by Lemma 21. When the β -process with

verifications terminates then this models assigning unique names by the algorithm. It follows that one iteration of the repeat-loop results in the algorithm terminating with proper names assigned with probability $1 - n^{-a}$. One iteration of the main repeat-loop does not result in termination with probability at most n^{-a} , so i iterations are not sufficient to terminate with probability at most n^{-ia} . This converges to 0 with increasing i so the algorithm terminates almost surely.

The performance metrics rely mostly on Lemma 20. We consider two cases, depending on which function $r(k)$ is used.

If $r(k) = k + 1$ then procedure GAUGE-SIZE-MC considers all the consecutive values of k up to $\lg n$, and for each such k , throwing a ball requires k random bits. We obtain that procedure GAUGE-SIZE-MC uses $\mathcal{O}(n \log^2 n)$ random bits. Similarly, to compute the number of selected values in an iteration of the main repeat-loop of this procedure takes time $\mathcal{O}(k)$, for the corresponding k , so this procedure takes $\mathcal{O}(\log^2 n)$ time. The value of **size** satisfies $\mathbf{size} \leq 2n$ with certainty. Therefore, $\mathcal{O}(n)$ memory registers are ever needed, while one throw of a ball uses $\mathcal{O}(\log n)$ random bits, after **size** has been computed. It follows that one iteration of the main repeat-loop of the algorithm, after procedure GAUGE-SIZE-MC has been completed, uses $\mathcal{O}(n \log n)$ random bits, by Lemmas 20 and 21, and takes $\mathcal{O}(\log n)$ time. Since one iteration of the main repeat-loop suffices with probability $1 - n^{-a}$, the overall time is dominated by the time performance of procedure GAUGE-SIZE-MC.

If $r(k) = 2k$ then procedure GAUGE-SIZE-MC considers all the consecutive powers of 2 as values of k up to $\lg n$, and for each such k , throwing a ball requires k random bits. Since the values k form a geometric progression, procedure GAUGE-SIZE-MC uses $\mathcal{O}(\log n)$ random bits per processor. Similarly, to compute the number of selected values in an iteration of the main repeat-loop of this procedure takes time $\mathcal{O}(k)$, for the corresponding k that increase geometrically, so this procedure takes $\mathcal{O}(\log n)$ time. The value of **size** satisfies $\mathbf{size} \leq 2n$ with certainty. By Lemma 20, $\mathcal{O}(n^2)$ memory registers are ever needed, so one throw of a ball uses $\mathcal{O}(\log n)$ random bits. One iteration of the main repeat-loop, after procedure GAUGE-SIZE-MC has been completed, uses $\mathcal{O}(n \log n)$ random bits, by Lemmas 20 and 21, and takes $\mathcal{O}(\log n)$ time. \square

The instantiations of algorithm COMMON-UNBOUNDED-MC are close to optimality with respect to some of the performance metrics we consider, depending on whether $r(k) = k + 1$ or $r(k) = 2k$. If $r(k) = k + 1$ then the algorithm's use of shared memory would be optimal if its time were $\mathcal{O}(\log n)$, by Theorem 2, but it misses space optimality by at most a logarithmic factor, since the algorithm's time is $\mathcal{O}(\log^2 n)$. Similarly, for this case of $r(k) = k + 1$, the number of random bits ever generated $\mathcal{O}(n \log^2 n)$ misses optimality by at most a logarithmic factor, by Proposition 1. In the other case of $r(k) = 2k$, the expected time $\mathcal{O}(\log n)$ is optimal, by Theorem 3, the expected number of random bits $\mathcal{O}(n \log n)$ is optimal, by Proposition 1, and the probability of error $n^{-\mathcal{O}(1)}$ is optimal, by Proposition 3, but the amount of used shared memory misses optimality by at most a polynomial factor, by Theorem 3.

12 Conclusion

We considered the naming problem for the anonymous synchronous PRAM when the number of processors n is known. We gave Las Vegas algorithms for four variants of the problem, which are determined by the suitable restrictions on concurrent writing and the amount of shared memory.

Each of these algorithms is provably optimal for its case with respect to the natural performance metrics such as expected time (as determined by the amount of shared memory) and expected number of used random bits.

We also considered four variants of the naming problem for an anonymous PRAM, when the number of processors n is unknown, and developed Monte Carlo naming algorithms for each of them. The two algorithms for a bounded number of shared registers are provably optimal with respect to the following three performance metrics: expected time, expected number of generated random bits and probability of error. It is an open problem to develop Monte Carlo algorithms for Arbitrary and Common PRAMs for the case when the amount of shared memory is unbounded, such that they are simultaneously asymptotically optimal with respect to these same three performance metrics: the expected time, the expected number of generated random bits and the probability of error.

The algorithms we gave cover the “boundary” cases of the model. One case is about a minimum amount of shared memory, that is, when only a constant number of shared memory cells are available. The other case is about a minimum expected running time, that is, when the expected running time is $\mathcal{O}(\log n)$; such performance requires a number of shared registers that grows unbounded with n . It would be interesting to have the results of this paper generalized by investigating naming on a PRAM when the number of processors and the number of shared registers are independent parameters of the model.

References

- [1] Y. Afek and Y. Matias. Elections in anonymous networks. *Information and Computation*, 113(2):312–330, 1994.
- [2] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, and R. Guerraoui. Tight bounds for asynchronous renaming. *Journal of the ACM*, 61(3):18:1–18:51, 2014.
- [3] D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu, and R. Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the 24th International Symposium on Distributed Computing (DISC)*, volume 6343 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2010.
- [4] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th ACM Symposium on Theory of Computing (STOC)*, pages 82–93, 1980.
- [5] D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. On the power of anonymous one-way communication. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, volume 3974 of *Lecture Notes in Computer Science*, pages 396–411. Springer, 2006.
- [6] D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing population protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4), 2008.
- [7] J. Aspnes, F. E. Fich, and E. Ruppert. Relationships between broadcast and shared memory in reliable anonymous distributed systems. *Distributed Computing*, 18(3):209–219, 2006.
- [8] J. Aspnes, G. Shah, and J. Shah. Wait-free consensus with infinite arrivals. In *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC)*, pages 524–533, 2002.

- [9] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.
- [10] H. Attiya and F. Ellen. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2014.
- [11] H. Attiya, A. Gorbach, and S. Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162–183, 2002.
- [12] H. Attiya and M. Snir. Better computing on the anonymous ring. *Journal of Algorithms*, 12(2):204–238, 1991.
- [13] H. Attiya, M. Snir, and M. K. Warmuth. Computing on an anonymous ring. *Journal of the ACM*, 35(4):845–875, 1988.
- [14] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley, 2nd edition, 2004.
- [15] P. Beame. Limits on the power of concurrent-write parallel machines. *Information and Computation*, 76(1):13–28, 1988.
- [16] P. Boldi and S. Vigna. An effective characterization of computability in anonymous networks. In *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, volume 2180 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2001.
- [17] F. Bonnet and M. Raynal. The price of anonymity: Optimal consensus despite asynchrony, crash, and anonymity. *ACM Transactions on Autonomous and Adaptive Systems*, 6(4):23, 2011.
- [18] H. Buhrman, A. Panconesi, R. Silvestri, and P. Vitanyi. On the importance of having an identity or, is consensus really universal? *Distributed Computing*, 18(3):167–176, 2006.
- [19] J. Chalopin, Y. Métivier, and T. Morsellino. Enumeration and leader election in partially anonymous and multi-hop broadcast networks. *Fundamenta Informatica*, 120(1):1–27, 2012.
- [20] B. S. Chlebus, G. De Marco, and M. Talo. Naming a channel with beeps. *CoRR*, abs/1608.04174, 2016.
- [21] B. S. Chlebus, K. Diks, and A. Pelc. Waking up an anonymous faulty network from a single source. In *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS)*, pages 187–193. IEEE, 1994.
- [22] B. S. Chlebus and D. R. Kowalski. Asynchronous exclusive selection. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 375–384, 2008.
- [23] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal of Computing*, 15(1):87–97, 1986.
- [24] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, 2nd edition, 2006.
- [25] D. Dereniowski and A. Pelc. Leader election for anonymous asynchronous agents in arbitrary networks. *Distributed Computing*, 27(1):21–38, 2014.

- [26] Y. Dieudonné and A. Pelc. Anonymous meeting in networks. *Algorithmica*, 74(2):908–946, 2016.
- [27] K. Diks, E. Kranakis, A. Malinowski, and A. Pelc. Anonymous wireless rings. *Theoretical Computer Science*, 145(1&2):95–109, 1995.
- [28] Ö. Egecioğlu and A. K. Singh. Naming symmetric processes using shared variables. *Distributed Computing*, 8(1):19–38, 1994.
- [29] Y. Emek, J. Seidel, and R. Wattenhofer. Computability in anonymous networks: Revocable vs. irrevocable outputs. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP), Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 183–195. Springer, 2014.
- [30] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume I. Wiley, 3rd edition, 1968.
- [31] F. E. Fich, F. Meyer auf der Heide, P. Ragde, and A. Wigderson. One, two, three...infinity: Lower bounds for parallel computation. In *Proceedings of the 17th ACM Symposium on Theory of Computing (STOC)*, pages 48–58, 1985.
- [32] F. E. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.
- [33] P. Flocchini, E. Kranakis, D. Krizanc, F. L. Luccio, and N. Santoro. Sorting and election in anonymous asynchronous rings. *Journal of Parallel and Distributed Computing*, 64(2):254–265, 2004.
- [34] P. Fraigniaud, A. Pelc, D. Peleg, and S. Perennes. Assigning labels in an unknown anonymous network with a leader. *Distributed Computing*, 14(3):163–183, 2001.
- [35] L. Gašieniec, E. Kranakis, D. Krizanc, and X. Zhang. Optimal memory rendezvous of anonymous mobile agents in a unidirectional ring. In *Proceedings of the 32nd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 3831 of *Lecture Notes in Computer Science*, pages 282–292. Springer, 2006.
- [36] C. Glacet, A. Miller, and A. Pelc. Time vs. information tradeoffs for leader election in anonymous trees. In *Proceedings of the 27th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 600–609. SIAM, 2016.
- [37] R. Guerraoui and E. Ruppert. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3):165–177, 2007.
- [38] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.
- [39] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
- [40] P. Jayanti and S. Toueg. Wakeup under read/write atomicity. In *Proceedings of the 4th International Workshop on Distributed Algorithms (WDAG)*, volume 486 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 1990.

- [41] J. Keller, C. W. Keßler, and J. L. Träff. *Practical PRAM Programming*. Wiley Series on Parallel and Distributed Computing. Wiley, 2001.
- [42] D. R. Kowalski and A. Malinowski. How to meet in anonymous network. *Theoretical Computer Science*, 399(1-2):141–156, 2008.
- [43] E. Kranakis and D. Krizanc. Distributed computing on anonymous hypercube networks. *Journal of Algorithms*, 23(1):32–50, 1997.
- [44] E. Kranakis, D. Krizanc, and F. L. Luccio. On recognizing a string on an anonymous ring. *Theory of Computing Systems*, 34(1):3–12, 2001.
- [45] E. Kranakis, D. Krizanc, and J. van den Berg. Computing boolean functions on anonymous networks. *Information and Computation*, 114(2):214–236, 1994.
- [46] E. Kranakis and N. Santoro. Distributed computing on oriented anonymous hypercubes with faulty components. *Distributed Computing*, 14(3):185–189, 2001.
- [47] S. Kutten, R. Ostrovsky, and B. Patt-Shamir. The Las-Vegas processor identity problem (How and when to be unique). *Journal of Algorithms*, 37(2):468–494, 2000.
- [48] R. J. Lipton and A. Park. The processor identity problem. *Information Processing Letters*, 36(2):91–94, 1990.
- [49] C. McDiarmid. On the method of bounded differences. In J. Siemons, editor, *Surveys in Combinatorics, 1989*, pages 148–188. Cambridge University Press, 1989.
- [50] Y. Métivier, J. M. Robson, and A. Zemmari. Analysis of fully distributed splitting and naming probabilistic procedures and applications. *Theoretical Computer Science*, 584:115–130, 2015.
- [51] O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Naming and counting in anonymous unknown dynamic networks. In *Proceedings of the 15th Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 8255 of *Lecture Notes in Computer Science*, pages 281–295. Springer, 2013.
- [52] M. Mitzenmacher and E. Upfal. *Probability and Computing*. Cambridge University Press, 2005.
- [53] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [54] A. Panconesi, M. Papatriantafilou, P. Tsigas, and P. M. B. Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3):113–124, 1998.
- [55] A. Pelc. Activating anonymous ad hoc radio networks. *Distributed Computing*, 19(5-6):361–371, 2007.
- [56] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.
- [57] E. Ruppert. The anonymous consensus hierarchy and naming problems. In *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS)*, volume 4878 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2007.

- [58] N. Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 173–179, 1999.
- [59] B. Schieber and M. Snir. Calling names on nameless networks. *Information and Computation*, 113(1):80–101, 1994.
- [60] P. van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. The MIT Press, 1990.
- [61] U. Vishkin. Using simple abstraction to reinvent computing for parallelism. *Communications of the ACM*, 54(1):75–85, 2011.
- [62] M. Yamashita and T. Kameda. Computing on anonymous networks: Part I-characterizing the solvable cases. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):69–89, 1996.
- [63] A. C. Yao. Probabilistic computations: Toward a unified measure of complexity. In *Proceedings of the 18th Symposium on Foundations of Computer Science (FOCS)*, pages 222–227. IEEE Computer Society, 1977.